目錄

Rails 5 开发进阶

本书主要包括两部分：Rails 源码剖析和 Rails 使用指南。Rails 是一个 Web 开发框架，也是一个工具。"工欲善其事必先利其器"，想要更好的使用 Rails 这个工具，清楚其背后的魔法，阅读源代码是必备功课。

本书尽量做到系统、全面，从源码出发，会讲解到原理。本书大部分内容为原创，少数内容为整理网上资料，鉴于参考资料太多，恕我不能一一列举来源。

本书适合什么样的读者？

- 想要更好的使用 Rails
- 准备阅读 Rails 的源代码
- 想知道 Rails 的整体架构
- 想清楚 Rails 背后的魔法

本书包含了哪些内容？

- 一些文档里找不到的方法
- 每个模块的关键所在
- 源码阅读路线图

本书不讲哪些内容？

- 安装、部署
- 奇技淫巧
- Ruby 语法、Rails 入门
- 和具体业务相关的问题或功能
- 非常具体、底层的代码实现

本书目的

- 让使用其它语言、框架的技术人员，能看懂是什么
- 让已经入门的 Rails 使用人员，知道更多原理，能看懂怎么用

联系我

邮箱：leekelby@gmail.com

2017-6-16 更新说明

ActiveModel validate 的使用； Turbolinks 5 的推荐。

Active Job

Web 开发过程中，会遇到很多需要异步处理的任务：发邮件、发短信、图片处理、下载东西、spam 检测等。一般来说，这样的程序执行时间长，并且不需要实时把结果反馈给用户。我们可以将这些处理时间长的请求剥离出来异步处理，并及时响应页面的请求。

市面上已经有不少和这有关的 gem，它们封装好了接口，我们直接调用即可，比如受到广泛使用的 Resque、Sidekiq、Delayed job 等。

这些 gem 做的事情都大同小异，但特性上却各自不同：

	异步	队列	延时(定时)	优先级	超时	重试
Delayed Job	是	是	是	任务	全局	全局
Resque	是	是	是 (Gem)	队列	全局	是
Sidekiq	是	是	是	队列	否	任务

并且，在实际使用过程中，你会关心性能怎么样、是否支持多线程、持久化方案，以及 API 使用难易程度、稳定性等问题。也就是说，你有可能要更改异步处理所使用的 gem.

必要的话，你要快速切换到另一种异步处理方案上。更进一步，你不想被绑在一颗树上，不想每换一种方案都学习其专有的语法。

Active Job 解决了这个问题，它统一了接口，做到了：不同的延迟任务，一样的 **API.** Active Job 本身并不提供真正的后端任务解决方案，它做的主要是适配，你可以选择使用自己需要的方案，并且以后可以迁移到其它方案。

Queue Adapter

定义：配置使用哪种后端任务、队列管理方式。默认使用的 queue_adapter 是 `:inline` ，处理方式是立即执行任务。你需要自己设置 queue_adapter.

```
ActiveJob::Base.queue_adapter = :inline
# 或类似
Rails.application.config.active_job.queue_adapter = :test
```

已经支持 Resque、Sidekiq、Delayed Job 等常用延迟任务 gem，所有可用 adapter:

```
:backburner, :delayed_job, :qu, :que, :queue_classic,
:resque, :sidekiq, :sneakers, :sucker_punch,
:inline, :test
```

Queue Name

定义：任务都是先进队列里，队列都有名字的，方便管理。默认使用的 queue_name 是 "default"

可以定制：

```
class MyJob < ActiveJob::Base
  queue_as :my_jobs

  def perform(record)
    # ...
  end
end
```

通过 `config.active_job.queue_name_prefix=` 可给所有队列名加前缀。

Queue Priority

定义：队列有优先级这个属性，优先级高的会被先执行。类方法 `queue_with_priority` 可以进行设置，对整个类有效：

```
class PublishToFeedJob < ActiveJob::Base
  queue_with_priority 50

  def perform(post)
    post.to_feed!
  end
end
```

可用实例方法 `priority` 获取，由上面统一设置的。

Core

调用：设置任务所在队列、队列优先级行。类方法 `set` 使用举例：

```
VideoJob.set(queue: :some_queue).perform_later(Video.last)
VideoJob.set(wait: 5.minutes).perform_later(Video.last)
VideoJob.set(wait_until: Time.now.tomorrow).perform_later(Video.
last)

VideoJob.set(queue: :some_queue, wait: 5.minutes)
  .perform_later(Video.last)

VideoJob.set(queue: :some_queue, wait_until: Time.now.tomorrow)
  .perform_later(Video.last)

VideoJob.set(queue: :some_queue, wait: 5.minutes, priority: 10)
  .perform_later(Video.last)
```

`set` 支持可选参数：:wait、:wait_until、:queue、:priority，它的具体实现由 ConfiguredJob 完成，主要是处理各个参数，起到配置作用。

> Note：可以不使用 `set` 直接调用 `perform_later` 进队列，然后等待执行。

类方法：

```
deserialize
```

实例方法：

```
serialize
deserialize
```

它们只是后端任务信息的一种方式，在此不必深究。

Enqueuing 入队与重试

调用。

常用方法：

```
enqueue
```

使用举例：

```
my_job_instance.enqueue

# 目前，只接受以下几种参数
my_job_instance.enqueue wait: 5.minutes
my_job_instance.enqueue queue: :important
my_job_instance.enqueue wait_until: Date.tomorrow.midnight
my_job_instance.enqueue priority: 10
# 若延迟 gem 本身不支持定时，会提示 wait、wait_until 不可用
```

入队列、执行任务失败，捕捉后，还可以重试：

```
retry_job
```

使用举例：

```
class SiteScrapperJob < ActiveJob::Base
  rescue_from(ErrorLoadingSite) do
    retry_job queue: :low_priority
  end

  def perform(*args)
    # raise ErrorLoadingSite if cannot scrape
  end
end
```

除上述两实例方法外，还有类方法：

```
perform_later
```

Execution 执行

调用。

```
# 实例方法
perform_now
execute # 由子类，也就是我们所定义的 Job 实现具体内容

# 类方法
perform_now # 简单封装了实例方法 perform_now

execute
```

使用举例：

```
MyJob.new(*args).perform_now

MyJob.perform_now("mike")
```

使用 `perform_now` 代码会立即执行，在这开发环境会很实用。

Callbacks 回调

定义 + 自动调用。

比某些延迟 gem 多做了一点点，除了队列&执行本身外，还可以有回调：

```
before_enqueue
around_enqueue
after_enqueue

before_perform
around_perform
after_perform
```

使用举例：

```
class VideoProcessJob < ActiveJob::Base
  queue_as :default

  after_perform do |job|
    UserMailer.notify_video_processed(job.arguments.first)
  end

  def perform(video_id)
    Video.find(video_id).process
  end
end
```

其它几个方法类似。

> Note: 实现上，使用了 ActiveSupport::Callbacks 的 define_callbacks、set_callback、run_callbacks 等方法。

提示

创建任务、进队列、执行任务这几个步骤，尽管我们可以区分开，但很多时候它们是交织在一起的(从 API 上就能看出)，我们可以不严格区分。

使用 Active Job 有利必有弊，可能面临以下问题：

原 gem 本身的特性没能充分发挥，灵活性降低，和其它 gem 的集成会变复杂。

其它多个类或模块，统一在此列举。

命令行快捷生成

```
rails generate job NAME [options]
```

异常捕获与处理

使用 Active Support 的异常捕获方法 `rescue_from`

```
class GuestsCleanupJob < ActiveJob::Base
  queue_as :default

  # 异常捕获
  rescue_from(ActiveRecord::RecordNotFound) do |exception|
   # 异常处理
  end

  def perform
    # ...
  end
end
```

参数支持 Global ID

一般入队列(enqueue_in、enqueue_at 和 enqueue) 只传能够标识对象的那部分参数(如：class、id)，出队列/执行的时候再根据这些参数获取对象。

但因为 serialize_argument 支持的类型有多种，其中就包括 GlobalID::Identification. 所以我们可以传递一个"活的对象"进队列，而不只是它的一部分(如：class、id).

使用 Global ID 前后对比。

之前：

```
class TrashableCleanupJob
  def perform(trashable_class, trashable_id, depth)
    # 出队列/执行的时候需要根据 trashable_class 和 trashable_id 查询
相应 trashable
    trashable = trashable_class.constantize.find(trashable_id)
    trashable.cleanup(depth)
  end
end
```

之后：

```
class TrashableCleanupJob
  def perform(trashable, depth)
    # 出队列/执行的时候直接使用 trashable
    trashable.cleanup(depth)
  end
end
```

> Note: 以前直接传递对象是不规范的写法。

获取任务的部分信息

所带参数、任务id、所在队列名、优先级等：

```
@arguments  = arguments
@job_id     = SecureRandom.uuid
@queue_name = self.class.queue_name
@priority   = self.class.priority
```

```
# Job arguments
attr_accessor :arguments

# Timestamp when the job should be performed
attr_accessor :scheduled_at

# Job Identifier
attr_accessor :job_id

# ID optionally provided by adapter
attr_accessor :provider_job_id

# I18n.locale to be used during the job.
attr_accessor :locale
```

Async Job

已经异步了，再给你一个选择：是否要并发？

```
Rails.application.config.active_job.queue_adapter = :async
```

进队列后，任务运用 Ruby 自身的并发。数据放在内存里，运行速度加快了；没有持久化，重启会丢失数据。

一般来说，生产环境你不能使用，但开发与测试或许可以体验。

Arguments 参数处理

接受的参数类型很广泛，需要先处理一下。

进队列时参数需要 `serialize` ，执行前参数需要 `deserialize`

当然，这都是自动完成的。

Queue Adapters

原来，不同的延迟任务 gem 有各自不同的 self.perform、perform、run、work，现在：
都有同名的 self.enqueue 和 self.enqueue_at

Logging

around_enqueue、around_perform 和 before_enqueue 有日志记录；
enqueue、enqueue_at、perform_start、perform 等过程也有日志记录。

默认和 Rails 应用使用同一日志系统。

Identifier

每个任务都有全局唯一的 job_id

Configured Job

可实例化自定义的 Job 类，然后调用 perform_now 或 perform_later. Core 里的 set 类方法会用到它，主要作用是处理各个参数，起到配置作用。

解析 queue_adapter 及其 API

queue_adapter 是 Delayed Job、Resque、Sidekiq 等不同的延迟任务抽象而来。

而 queue_adapter 所用的 API(enqueue_at、enqueue_in、enqueue 等)，也是从原延迟任务所提供的 API 抽象而来。

弊端及混合使用注意事项

它是对 Resque、Sidekiq 等的封装，对比其文档，可以看出部分功能牺牲掉，部分特性被砍掉了。

当你想要这部分功能、特性，而又同时使用着 Active Job 则需注意：

1）这些底层的 gem 传递的是 id，而 Active Job 因为有 GlobalID 支持可以传 record 对象。

2）部分功能、特性 Active Job 是没有的，定义、调用的时候需要特别注意，因为有的地方即使你用错了，它也不会报错。

Action Mailer

Action Mailer 是 Rails 内建的组件，用来处理邮件相关业务。

它依赖于 Rails 内建的其它组件，如：Active Job、Abstract Controller 和 Action View，以及外部 gem 'mail'.

因为是 Rails 内建的组件，所以使用上通常集成于 Rails 项目，但其实它也可以在 Rails 之外使用。

核心是 gem 'mail'

既然是用来处理邮件相关业务，那么核心功能当然就是邮件处理，这部分由 gem 'mail' 完成。

gem 'mail' 可用于邮件处理，包括创建、发送、和接收等，示例如下:

```ruby
require 'mail'

Mail.defaults do
  delivery_method :smtp,
    :address              => "smtp.gmail.com",
    :port                 => 587,
    :domain               => "example.com",
    :authentication       => :plain,
    :user_name            => "<gmail_username>@gmail.com",
    :password             => "<gmail_password>",
    :enable_starttls_auto => true
end

mail = Mail.new do
  from     'from@example.com'
  to       'to@example.org'
  subject  'First multipart email sent with Mail'

  text_part do
    body 'This is plain text'
  end

  html_part do
    content_type 'text/html; charset=UTF-8'
    body '<h1>This is HTML</h1>'
  end
end

mail.deliver!
```

上面的例子使用了 Gmail 做为邮件服务器，所以需要用到 Gmail 用户名和密码，实际应用中你可以在本地搭建或使用其它第三方邮件服务器。更多示例，可以参考 mail#usage

> Note: 发送邮件，还可以使用标准库 Net::SMTP

引入其它，为了更好用

直接使用 gem 'mail'，创建、发送邮件等最最基本的功能是实现了，但并不好用。缺少灵活的配置，内容与模板没有分离等。Action Mailer 改善了它们，举例单独使用 action_mailer：

```
# mailer.rb
require 'action_mailer'

# 邮件发送报错时，是否把错误信息发送给用户。开发环境下，可设置为 true
ActionMailer::Base.raise_delivery_errors = true

# 使用单独配置文件的话，可以用 config.action_mailer 代替下面的 ActionM
ailer::Base
ActionMailer::Base.delivery_method = :smtp
ActionMailer::Base.smtp_settings = {
  :address => "smtp.gmail.com",
  :port    => 587,
  :domain  => "example.com",
  :authentication => :plain,
  :user_name      => "test@example.com",
  :password       => "passw0rd",
  :enable_starttls_auto => true
}

ActionMailer::Base.view_paths = File.dirname(__FILE__)

class Mailer < ActionMailer::Base
  def daily_email
    @var = "var"

    # 发件人是这里的 from，不是上面 smtp_settings 里设置的
    mail(to: "to@gmail.com", from: "test@example.com", subject:
"test") do |format|
      format.text
      format.html
    end
  end
end

email = Mailer.daily_email
email.deliver_now
```

```
# mailer/daily_email.html.erb
<p>this is an html email</p>
<p> and this is a variable <%= @var %></p>
```

```
# mailer/daily_email.text.erb
this is a text email
and this is a variable <%= @var %>
```

配置部分可以抽取出来，模板和内容可以分开管理，创建和发送邮件也更加直观。

引入其它，为了更实用

通常，除了邮件发送外，我们还需要其它功能，这就有一个"集成"的过程。我们希望能够测试、能够自动化、能够预览邮件等。我们想用 Rails 其它组件提供的 render, before_action, url_for 以及丰富的 helper 方法。

运行下面命令即可自动生成邮件模板：

```
rails generate mailer Notifier welcome
```

实现方式：汝果欲学诗，功夫在诗外

Action Mailer 本身并没有"实现"什么功能。最基本，也是最核心的部分由 gem 'mail' 完成；为了更好用、更实用，加载或封装了 Active Job、Action Pack 和 Action View.

为了做好、做得上层次，下面是在邮件处理外，Action Mailer "实现"的一些功能：

- 集成现有服务(有时我们要的不仅仅只是邮件服务)
- 拦截器、观察者(发送之前、之后想做点什么？)
- 测试
- 日志记录
- 延迟发送
- 灵活的配置
- 内容与模板分离
- 丰富实用的 helper 方法

- 邮件预览
- 自动化(如：rails g mailer)

Base

它是 Action Mailer 里其它模块的集合中心（其它模块不直接对外提供接口，而是通过 Base 完成）。
同时，它还提供一些对外的接口，供我们直接使用。
此外，它还起到承上启下的作用，虽然我们可能感受不到。例如：与底层其它模块交互，负责将请求转向具体的 Mailer#action 等。

它是我们自定义 Mailer 的父类，是连接我们应用与 Action Mailer 的纽带，是 Action Mailer 连接 Abstract Controller 的纽带。

作用

它继承于 AbstractController::Base，包含了一些自身及 Abstract Controller 的模块(尽管有的模块它并没有使用到)，作用是为了让它的子类(我们自定义的 Mailer 类)能够"更好用、更实用"。

```
      YourMailer
          |
          V
   ActionMailer::Base
          |
          V
 AbstractController::Base
```

包含内容

包含了一些默认配置 default_params.

注册拦截器(interceptor)或观察者(observer).

请求到邮件处理 process.

创建邮件实例 (因为 Base include 了多个模块，所以这个实例可以使用这些模块所定义的方法)。

邮件(mail)、附件(attachments)

`mail(headers = {}, &block)` 表示邮件对象。
可接收一个代码块做为参数，header(头部)可接受：

参数	含义
:subject	主题
:from	发件人
:to	收件人
:cc	抄送
:bcc	密送
:reply_to	回邮地址
:date	时间

> Note: 想了解更多 header 信息，可以点击 Email#Header_fields

使用举例：

```
def welcome(user)
  @user = user

  mail(to: @user.email, subject: 'Welcome to My Awesome Site')
end
```

`attachments()` 允许你在邮件里添加附件。

类型：

```
a_mailer.attachments.class
# => Mail::AttachmentsList

# 类似数组
a_mailer.attachments
# => []
```

使用举例：

```
# 文件名的方式
mail.attachments['filename.jpg'] # => Mail::Part 实例对象或 nil

# 索引的方式
mail.attachments[0]                # => Mail::Part (第一个实例对象)
```

只需要指定文件名、文件类型，然后 Rails 会自动帮你计算出 Content-Type, Content-Disposition, Content-Transfer-Encoding 和 base64 编码等附件内容。

你也可以重写它们：

```
mail.attachments['filename.jpg'] = { mime_type: 'application/x-gzip',
                                      content: File.read('/path/to/filename.jpg') }
```

设置默认值(default & default_options=)

`default(value = nil)` 是ActionMailer::Base提供的方法，用来设置 default_params，默认已经有：

```
default_params = {
  mime_version: "1.0",
  charset:      "UTF-8",
  content_type: "text/plain",
  parts_order:  [ "text/plain", "text/enriched", "text/html" ]
}
```

我们可以配置(value 必需是 Hash 类型)：

```
# 下面两者一样
config.action_mailer.default { from: "no-reply@example.org" }
config.action_mailer.default_options = { from: "no-reply@example
.org" }

# 常见配置(开发模式下)
config.action_mailer.default_url_options = { :host => 'localhost
:3000' }

# 配置邮件发送方式
config.action_mailer.delivery_method = :smtp

# 根据不同发送方式，做不同配置
config.action_mailer.smtp_settings = {
  address: 'smtp.gmail.com',
  port: 587,
  domain: 'your_domain.com',
  # 完整的邮箱地址
  user_name: 'your_email@gmail.com',
  password: 'your_gmail_password',
  authentication: 'plain',
  enable_starttls_auto: true
}

# 邮件发送报错时(比如：邮箱错了)，是否抛异常。开发环境下，可设置为 true
config.action_mailer.raise_delivery_errors = true

# 是否真的发邮件，默认已经是 true
config.action_mailer.perform_deliveries = true
```

```
# 遇到以下报错，可考虑：
# Net::SMTPAuthenticationError: 535 #5.7.0 Authentication failed

:openssl_verify_mode => 'none'
```

它和配置文件里的 `default_options=` 是一样。

在 config/environments/ 目录下针对不同执行环境会有不同的邮件服务器设置：

```
config.action_mailer.default_options = { from: "no-reply@example
.org" }
```

前者对当前 Controller 及其子类有效，而后者对当前环境下所有 Controller 有效。除了使用的地方不同，导致作用域稍有不同外，两者本质是一样的。

设置头部消息 headers

使用 `headers` 可以设置邮件的头部消息，例如：

```
headers['X-Special-Domain-Specific-Header'] = "SecretValue"

headers 'X-Special-Domain-Specific-Header' => "SecretValue",
        'In-Reply-To' => incoming.message_id
```

直接调用了 Mail::Message#headers 方法，默认已经有选项：

```
:subject
:sender
:from
:to
:cc
:bcc
:reply-to
:orig-date
:message-id
:references
```

接收邮件 receive

Rails 处理邮件，不常用，而且会比较耗费资源，所以不推荐。但如果你要用的话，你可以实现 `receive(raw_mail)` 方，唯一的参数就是，接收到的邮件内容。
Rails 会先创建对应的 Mail 邮件对象，之后才进行后续处理。

创建邮件对象时的魔法

细心的你应该发现，我们在 Mailer 类里定义的是实例方法，但创建 mailer 对象用的却是类方法。

这里隐藏着魔法，当找不到此类方法时，就会调用 Rails 重新实现的 `method_missing` 类方法, 会先检查 action_methods 里是否有同名方法，如果有，则(把此方法当做参数)创建 MessageDelivery 对象。

其它

类方法：

```
register_interceptor  # 简单封装 Mail.register_interceptor
register_interceptors # 简单封装上面的 register_interceptor, 可注册多个拦截器。

register_observer     # 简单封装 Mail.register_observer
register_observers    # 简单封装上面的 register_observer, 可注册多个观察者。
```

方法：

```
mailer_name & controller_path
```

返回 Mailer 类的名字，默认与类名同名。

拦截器 register_interceptor

邮件真正地发送之前想做点什么？-- 使用拦截器。

为了"开发和测试尽量的接近生产环境"和"知道发送出去的邮件真正的样子"等目的，我们希望在非生产环境下，能够查看邮件发送情况。

解决方案：

- (开发、测试)用邮件预览 gem

我们可以使用 mailcatcher 或 letter_opener 等 gem 实现。

- (生产)只发送到特定的邮箱

比如，我们注册一些特殊账号或邮箱，然后发送给我们自己。

- 用拦截器

也就是下面要介绍的。

注册拦截器。举例一(可指定条件)：

```
if Rails.env.development?
  ActionMailer::Base.register_interceptor(DevelopmentMailInterce
ptor)
end
```

注册拦截器。举例二(没有加载的话，需要先加载)：

```
require 'whitelist_interceptor'
ActionMailer::Base.register_interceptor(WhitelistInterceptor)
```

注册拦截器。举例三(没有加载的话，需要先加载)：

```
require 'environment_interceptor'
ActionMailer::Base.register_interceptor(EnvironmentInterceptor)
```

> 上述代码，一般放在 config/application.rb 或 config/environments/file_name.rb 或 config/initializers/file_name.rb 文件里。

实现 `delivering_email`. 举例一（在最后时刻替换掉要发送到的邮箱）：

```
class DevelopmentMailInterceptor
  def self.delivering_email message
    message.subject = "[#{message.to}] #{message.subject}"
    message.to = "overwrite_to@example.com"
  end
end
```

实现 `delivering_email`. 举例二（白名单，群发公司邮箱）：

```
class WhitelistInterceptor
  def self.delivering_email message
    unless message.to.join(' ') =~ /(@yourcompany.com)/i
      message.subject = "#{message.to} #{message.subject}"
      message.to = ENV['NOTIFICATIONS_EMAIL']
    end
  end
end
```

实现 `delivering_email`. 举例三（非生产环境时，标明发邮件所在的运行环境）：

```
class EnvironmentInterceptor
  def self.delivering_email message
    unless Rails.env.production?
      message.subject = "[#{Rails.env.capitalize}] #{message.sub
ject}"
    end
  end
end
```

> 上述代码，一般放在 lib/file_name.rb 文件里。

上面大概思路已经给出，实际项目如何使用可以自行决定。

订阅者 register_observer

邮件发送之后想做点什么？-- 使用观察者。

类似 register_interceptor，使用观察者：

```
class MailObserver
  # 实现 delivering_email 方法
  def self.delivered_email message
    # ...
  end
end

ActionMailer::Base.register_observer(MailObserver)
```

注意：这里实现的是 `delivered_email` 方法，而之前实现的是 `delivering_email` 方法。

Mail Helper

几个邮件相关的 **Helper** 方法。

如：attachments、mailer、message，及不太常用的 block_format、format_paragraph.

方法	解释
message()	表示邮件对象，即 Mail::Message 实例对象
attachments()	表示邮件里面的附件
format_paragraph(text, len = 72, indent = 2)	格式化文字内容。默认行首空两格，每行长度不超过 72 个字符
block_format(text)	使用上面的 format_paragraph 来处理大段的文字内容
mailer()	YourMailer 的实例对象，类似 YourController 的实例对象

另，Mailer 渲染视图和 Controller 渲染视图，它们原理和过程基本上是一致的。所以除了上述 Helper 方法外，Action View 提供的辅助方法在这里也可用。

Message Delivery

应用层面的邮件发送。

发送邮件时，会调用 YourMailer#action，在这里就会创建 Message Delivery 实例对象。

实例方法：

```
deliver_now
deliver_now!

deliver_later
deliver_later!
```

和

```
message
```

`deliver_now` 和 `deliver_now!` 立即发送邮件。
`deliver_later` 和 `deliver_later!` 延迟发送邮件，可接受参数 :wait、:wait_until 或 :queue.

通常，我们都是创建邮件并发送：

```
Notifier.welcome("helloworld@example.com").deliver_now
```

也可以先创建邮件对象，稍后邮件对象调用方法发送邮件：

```
message = Notifier.welcome("helloworld@example.com")
# => 得到 ActionMailer::MessageDeliver 实例对象

message.deliver_now
# 执行发送邮件
```

无论哪种方式发送邮件，都要先得到 message 对象，调用 deliver 方法，然后调用 gem 'mail' 相关代码实现发送。

获取 Mail::Message 实例对象(deliver 操作其实是由它发出)：

```
# 表示邮件对象，它本质是 Mail::Message 的实例对象
message = Notifier.welcome(david).message
```

其它使用示例：

```
Notifier.welcome(User.first).deliver_later
Notifier.welcome(User.first).deliver_later(wait: 1.hour)
Notifier.welcome(User.first).deliver_later(wait_until: 10.hours.from_now)
```

> Note: 这里的邮件"发送"，指的是我们应用层面的"发送"。至于 Rails 底层如何实现邮件"发送"，参考【Delivery Methods】章节。

其它

更多关于 Action Mailer

Action Mailer 使用模板来创建邮件与 Action Controller 使用模板渲染视图，原理类似。并且，渲染过程都会运用到 Action View 的 Rendering 模块。

Action Mailer 提供我们 Mailer 类和视图。Mailer 类放在 app/mailers 目录下，关联的视图文件在 app/views 目录下。

ActionMailer::Base 继承于类 AbstractController::Base, 又包含但不限于以下'外部'模块，根据 Ruby 规则，它们也是可调用的。包括：

模块	作用
ActionMailer::DeliveryMethods	邮件发送
ActionMailer::Previews	邮件预览
AbstractController::Rendering	渲染

AbstractController::Helpers 引进、输出辅助方法 AbstractController::Translation | I18n 相关 AbstractController::Callbacks | 支持回调 ActionView::Layouts | 视图布局等

`C < B < A`

有时候 B 要 extend A，并不是为了 B 自己使用，而是为了方便 C. ActionMailer::Base 部分代码充当的角色和 B 一样，它自己却没有使用，而是为了方便我们自定义的 Mailer 类调用。

如：

Mailer 里：

```
before_action :add_inline_attachment!

layout "mailer"

helper :application
```

View 里：

```
<%= url_for(host: "example.com", controller: "welcome", action:
"greeting") %>

<%= users_url(host: "example.com") %>
```

另，邮件是要发送出去给别人看的，所以邮件里的 url 不支持使用相对路径。

快速生成 Mailer 和模板

通过以下命令创建 mailer 类和视图：

```
$ rails g mailer UserMailer welcome

create   app/mailers/user_mailer.rb
invoke   erb
create     app/views/user_mailer
create     app/views/user_mailer/welcome.text.erb
invoke   test_unit
create     test/mailers/user_mailer_test.rb
```

~~Inline Preview Interceptor~~

Rails 自带的拦截器，可以将邮件里的图片 src 转换 data，以方便显示。
原因有多个，其中之一：有的邮件客户端会过滤图片的，通过转换可以提高图片的显示概率。

默认已经使用此拦截器，不想使用的话，需要手动删除：

```
ActionMailer::Base.preview_interceptors.delete(ActionMailer::Inl
inePreviewInterceptor)
```

~~Collector~~

mail 方法可以接收一个代码块，你可以在这里指定渲染模板的格式及内容等：

```
mail(to: user.email) do |format|
  format.text
  format.html
end

# 或

mail(to: user.email) do |format|
  format.text { render plain: "Hello World!" }
  format.html { render html: "<h1>Hello World!</h1>".html_safe }
end
```

代码块里的 `format` 即为 Collector 的实例对象。

也只有在这个时候，这里的 Collector 才有用到。它和 AbstractController::Collector Mime 相关。

> Note: 默认会发送和 mail 所在方法名同名的所有模板，不区分 Mime 格式。这也是我们常用的。

~~Delivery Job~~

使用 Active Job，配置以便延迟发送邮件。

Delivery Methods 定制与新增

底层选择邮件"发送"方式。

Rails 本身没有实现此功能，但提供了几种方式供我们选择。

调用 Rails 已有发送程序

delivery_method 默认已经有：

```
:smtp     => Mail::SMTP
:file     => Mail::FileDelivery
:sendmail => Mail::Sendmail
:test     => Mail::TestMailer
```

我们可以根据需求，选择、配置、使用它们。

使用已有邮件服务

- 如使用 gem 'letter_opener' 直接配置：

```
# config/environments/development.rb
config.action_mailer.delivery_method = :letter_opener
```

- 如使用 gem 'aws-ses' 配置：

```
ActionMailer::Base.add_delivery_method :ses, AWS::SES::Base,
  :access_key_id     => 'abc',
  :secret_access_key => '123'
```

然后：

```
config.action_mailer.delivery_method = :ses
```

"新增"邮件服务

完全的"新增"邮件服务，难度上比较大。比较实际的做法是"继承"于已有的实现，并且在 Rails 里注册一下，然后使用。

1) "继承"于已有的实现(这里是 SMTP)：

```
require 'mail'

class CustomSmtpDelivery < ::Mail::SMTP
  # SMTP 配置
  def initialize(values)
    self.settings = {:address => "smtp.gmail.com",
                     :port => 587,
                     :domain => 'yourdomain',
                     :user_name => "gmail_username",
                     :password => "gmail_password",
                     :authentication => 'plain',
                     :enable_starttls_auto => true,
                     :openssl_verify_mode => nil
                    }.merge!(values)
  end

  # 或者，把 SMTP 配置放到配置文件里
  def initialize(options = {})
    self.settings = options
  end

  attr_accessor :settings

  # 必须重新实现 deliver! 方法
  def deliver!(mail)
    # 把收箱人替换成指定的，这里功能类似拦截器
    mail['to'] = "to@example.com"
    mail['bcc'] = []
    mail['cc'] = []
    super(mail)
  end
end
```

2) 在 Rails 里注册一下：

```
add_delivery_method :custom_smtp_delivery, CustomSmtpDelivery
```

另，自带的 smtp、file、sendmail 及 test 已经注册。

3) 配置使用刚才新增的 delivery method：

```
# config/environments/development.rb
config.action_mailer.delivery_method = :custom_smtp_delivery
```

在之后配置就变成了：

```
ActionMailer::Base.custom_smtp_delivery_settings = {
  :address => "smtp.gmail.com",
  :port => 587,
  # ...
}
```

Rails 已有发送程序及 gem 'letter_opener' 都是用这种方式实现，之后提供给我们使用的。

其它内容

提供类方法：

```
# 获取所有可用的 delivery_methods
class_attribute :delivery_methods

module ClassMethods
  # 必须结合 delivery_method :test 使用，存放着已经 deliver 的邮件对象，测试的时候可用到它。
  delegate :deliveries, :deliveries=, to: Mail::TestMailer
end
```

Previews & Preview

邮件预览相关。

Previews，主要是对外的接口，对于普通开发者来说主要是配置：

```
# 配置预览文件存放的位置，默认如下:
config.action_mailer.preview_path = "#{Rails.root}/lib/mailer_previews"

# 配置是否允许邮件预览。开发模式下，默认为 true
config.action_mailer.show_previews = true
```

默认可以到以下 url, 查看预览邮件：

```
http://localhost:3000/rails/mailers/
```

提供类方法：

```
register_preview_interceptor
register_preview_interceptors
```

Preview，主要是对内的实现，是我们自定义 YourPreview 的父类，提供一些普通 Web 开发者察觉不到的方法，如：

`preview_name` 返回自定义类名，但把 "Preview" 后缀去掉。如 YourPreview 返回 "Your"

`emails` 返回所有可预览的邮件

这里的预览，和 Mail Catcher、Letter Opener 等提供的预览不同，它属于规范的测试，而后者更类似于人肉测试。

> Note: 邮件预览，在 Rails 里也遵守 MVC. M 是 ActionMailer::Preview，V 是 rails/mailers/，C 是 Rails::MailersController

~~提供类方法：~~

```
all
emails

preview_name

call
email_exists?

find
exists?
```

相关使用，可以参考官方文档。

DeliveryJob

Action Mailer 默认就已经支持异步发送，因为它使用了 Rails 自带的 ActiveJob.

具体实现体现在这个 class 上，它完成了使用 `queue_as` 指定队列名，使用 `rescue_from` 异常报错，及 perform 等工作（尽管 perform 还是调用具体 Mailer 里的代码）。

Collector

支持响应不同格式的邮件模板，就和 Controller#action 里我们编写的类似：

```
mail(to: 'mikel@test.lindsaar.net') do |format|
  format.text
  format.html
end
```

Abstract Controller

Abstract Controller 做的事情很有限，但却很无私。

无论是它自己定义的方法，还是封装 Action View 和 Action Dispatch 得到的方法，最终都提供给 Action Controller 和 Action Mailer 使用。

这些方法(或模块)包含但不限于渲染(模板或局部模板)、Helpers、Callbacks、Mime、Url For 等。

它服务于 Action Controller 和 Action Mailer.

- 辅助 ActionController::Base 将战场转移到具体的 Controller#action(经过 Metal).
- 辅助 ActionMailer::Base 将战场转移到具体的 Mailer#action.

准备及转移工作在之前已经完成了，它只是在最后一步，调用具体 action.

Base

Action Dispatch 里转发先到 ActionController::Metal::action, 然后到 ActionController::Metal#dispatch, 接着到 AbstractController::Base#process 也就是这里的 `process` 方法，然后到 `process_action` 和 `send_action & send` ，最后到具体的 action 进行处理。

此外，还有一些平时用得不多，但比较有趣的方法。

重要的实例方法有：

```
process
```

重要的私有方法有：

```
process_action

send_action & send
```

重要的类方法有：

```
abstract!

action_methods
```

`action_methods` 返回当前类所包含的 action，默认等同于 public_instance_methods. 这里的类可以是 Controller，也可以是 Mailer. 对于 Abstract Controller 来说，它们都是 AbstractController::Base 的子类，概念一样。

其它实例方法：

```
controller_path

action_methods

available_action?
```

其它类方法：

```
clear_action_methods!

controller_path

hidden_actions

internal_methods

method_added(name)

supports_path?
```

`controller_path` 返回当前 Controller 所在的路径(包括目录、文件名)。例如，YourApp::PostsController 返回"your_app/posts".

其它私有方法：

```
action_method?

method_for_action
```

Action Mailer 和 Action Controller 都继承于 Abstract Controller：

```
ActionMailer::Base < AbstractController::Base

ActionController::Base < ActionController::Metal < AbstractController::Base
```

获取所有的 Controller 和 action

如何获取当前程序所有的 Controller 和 action？

获取所有的 Controller:

```
ApplicationController.descendants
```

开发环境默认是延迟加载的，所以获取的只是当前 Controller，为了达到目的，可以先运行以下命令：

```
Rails.application.eager_load!
```

生产环境，不必运行。

已经获取了所有的 Controller，那么获取某个 Controller 所有的 action，用 `action_methods` 方法即可：

```
PostsController.action_methods
```

Helpers

导出、引入辅助方法。

方法	解释
helper	把 Helper 方法变成 Controller 方法，针对的是整个 module
helper_method	把 Controller 方法变成 Helper 方法，针对的是单个 method

helper(*args, &block)

功能和 `include` 类似，只是稍微智能了一点。

参数类型，可分为 3 类：String、Symbol，Module，block。这些参数还可以混合使用。

- 当参数是模块名时，直接 include 此模块 (没有 requires)

```
helper FooHelper # => includes FooHelper
```

- 而当参数是字符串或符号时，会根据约定 require 相关文件，并 include 相关模块。

```
helper :foo
# => requires 'foo_helper' 和 includes FooHelper

helper 'resources/foo'
# => requires 'resources/foo_helper' 和 includes Resources::FooH
elper
```

此外，helper 可以接受并处理一个代码块。(不推荐)

最后要说的是：上述参数类型可以混合使用，你可以同时传递符号、字符串、模块和代码块给 helper 方法。

```
helper(:three, BlindHelper) { def mice() 'mice' end }
```

helper_method(*meths)

以元编程的形式定义同名 Helper 方法，然后 send 调用原 Controller 里的方法。

如下文举例，Controller 里有 `current_user` 方法，可以在视图里使用：

```
class ApplicationController < ActionController::Base
  helper_method :current_user, :logged_in?

  def current_user
    @current_user ||= User.find_by(id: session[:user])
  end

  def logged_in?
    current_user != nil
  end
end
```

在视图里:

```
<% if logged_in? %>Welcome, <%= current_user.name %><% end %>
```

helper 和 helper_method 可以简单理解为一对作用相反的操作。

除上述两方法外，还有：

```
modules_for_helpers
```

`modules_for_helpers` 被上面的 helper 方法所调用。

```
clear_helpers
```

`clear_helpers` 只保留与此 Controller 同名的 Helper 模块的辅助方法，其它模块的辅助方法清除掉。

Callbacks

Controller 里可用的回调：

```
before_action & append_before_action
prepend_before_action
skip_before_action

after_action & append_after_action
prepend_after_action
skip_after_action

around_action & append_around_action
prepend_around_action
skip_around_action

skip_action_callback & skip_filter # 将被废除
```

这些回调方法参照物都是 `process_action` 方法。

```
set_callback(:process_action, callback, name, options)

set_callback(:process_action, callback, name, options.merge(:pre
pend => true))

skip_callback(:process_action, callback, name, options)
```

具体实现：调用了 ActiveSupport::Callbacks 里的 `define_callbacks` 、 `set_callback` 、 `run_callback` 和 `skip_callbacks` 方法。

另：skip_action_callback 意味着 skip before、after 和 around.

Rendering

Controller 和 Mailer "渲染"的入口，并负责转递实例变量。

渲染，Abstact Controller 做的只是最外层的封装。具体由 Action View 的 Rendering 相关模块完成，然后给 Action Controller 和 Active Mailer 使用。

```
Action Controller & Active Mailer
                 |
                 V
         Abstact Controller
                 |
                 V
           Action View
```

它封装了 Action View 里的 Template Renderer 和 Partial Renderer，提供 render 给 Action Controller, Action Mailer 渲染模板文件或局部模板。

我们在 Controller#actions 里使用的 `render` 就是在这里定义的（尽管它只是简单调用，几乎没做任何事）。

所以，在使用 `render` 过程中有疑问的话，可以查看 View 里对应 render 方法的文档。

除上述方法外，还有：

```
view_assigns
```

`view_assigns` 通过它可以查看 Controller 和 Mailer 传递给 View 的实例变量及其内容。在 ActionView::Rendering 里被调用。(通过 Ruby 自带的 instance_variables、instance_variable_get 得到)

```
render_to_body
render_to_string
```

```
rendered_format
```

这些方法主要起到接口的作用，具体实现在上游或下游。

render 参数汇总与详解

参数主要有 3 类：渲染格式、指定内容、传递变量。

ActionController::Rendering

```
# 渲染普通文本，不带格式
:plain
# 举例 render plain: "OK"
```

```
# 渲染 html 格式的内容。不推荐使用，可用 html.erb 代替
:html
# 举例 render html: "<strong>Not Found</strong>".html_safe

# 渲染普通文本，不带格式。不推荐使用，可用 :plain 代替
:text

# 兼容废除的 :text，默认是普通文本。不推荐使用
:body

# 不渲染任何东西，已废除。不推荐使用，可用 :plain 代替
:nothing
```

> Note：不想渲染任何东西，还可以使用方法 `head` 或 `render_to_string`

```
# 指定 Header status
:status
# 指定 Header content_type
:content_type
# 指定 Header location
:location
```

```
# 必需与 block 结合使用，里面可以放 Prototype 等代码。不推荐使用
:update
```

AbstractController::Rendering

```
# 指定优先渲染的变种
:variant
```

例如，渲染 show action，但 `request.variant = :phone` 则优先渲染：

```
show.html+phone.erb
```

ActionView::Rendering

```
# 指定优先渲染的变种
:variant
# 指定渲染的格式
:formats

# 指定要渲染的模板
:template
# 指定要渲染的 action，它会重新确认对应的模板
:action
# 指定要渲染的局部模板
:partial
# 指定要渲染的文件
:file

# 渲染路径前面部分
:prefixes
# 举例 render :show, prefixes: 'posts'
# 类似 render "posts/show"
```

ActionView::Renderer

```
# 指定要渲染的局部模板
:partial
```

这里的 render 方法，是 AV and AC 的主要入口，如果有 `:partial` 则渲染的是局部模板；否则，渲染的是普通模板。

ActionView::PartialRenderer

AbstractRenderer 的子类之一。

```
# 配合 :collection，渲染局部模板时，各个局部模板之间穿插的内容
:spacer_template

# 布局
:layout

# 传递变量
:locals

# 指定渲染格式
:formats

# 指定要渲染的局部模板
:partial
```

渲染"局部模板"时，约定：每一个局部模板都有一个与之同名的"局部变量"。

例如：

```
render :partial => "person"
```

则在局部模板内有 `person` 可用，它代表什么内容 ，以及如何更改名字。和以下几个参数有关：

```
# 传递单个对象时，与局部模板同名的变量
:object
# 传递的对象别名，更改默认的局部变量名字
:as
# 传递的集合，每一项可用局部变量代替
:collection
```

> Note：前面提到的参数 `:locals` 也可传递变量，它们本质是一样的。

ActionView::TemplateRenderer

AbstractRenderer 的子类之一。

```
# 渲染普通文本
:plain

# 渲染 html 格式的内容
:html

# 渲染文件，一般是可下载的文件(不使用 layout)
# 你可以手动指定，例如 layout: true
:file

# 直接渲染代码(不使用 layout)，默认使用 ERB 格式。不推荐使用
:inline
# 举例 render inline: "<% products.each do |p| %><p><%= p.name %></p><% end %>"
# 必须配合 :inline，指定要渲染代码的类型
:type
# 举例 type: :builder

# 指定模板
:template

# 指定布局
:layout

# 传递变量
:locals
```

```
# 兼容 :text，默认是普通文本。不推荐使用
:body
# 渲染普通文本。不推荐使用
:text
```

ActionView::Helpers::RenderingHelper

View 里 render 方法所在地，对外提供接口，处于最外层。

```
# 指定局部模板
:partial
# 指定布局
:layout

# 传递变量
:locals
```

作用：根据参数是不是 Hash 类型，参数有没有带 block，决定了如何渲染。
如果不传递 hash，则默认渲染 partial 并且第 2 个及之后的参数做为 locals hash.

ActionController::Renderers (Metal 增强模块)

```
# 渲染内容为 JSON 格式。可用 json.jbuilder 代替
:json
# 举例 render json: @product
# 渲染内容为 JSON 格式时，额外提供的参数
:callback

# 渲染内容为 JS 格式。可用 js.erb 代替
:js
# 举例 render js: "alert('Hello Rails');"

# 渲染内容为 XML 格式
:xml
举例 render xml: @product
```

ActionController::Renderer

直接渲染模板，不依赖于具体 Controller#action 时，可额外传递的参数：

```
# 指定请求是否 HTTPS
:https

# 指定请求方法
:method
```

更多关于渲染...

render 在 Controller 和 View 的区别？

Controller 里默认渲染的是完整的模板(template)

走的路是 ActionController::Rendering -> AbstractController::Rendering -> ActionView::Rendering -> ActionView::Renderer#render

View 里默认渲染的是局部模板(partial)

走的路是 ActionView::Helpers::RenderingHelper#render -> ActionView::Renderer

通过上面的路径和特别指出的两个 render 方法里面的逻辑，不难看出为什么可以默认渲染 template 或 partial.

Controller 里的 render 是为了返回 self.response_body 而 View 里的 render 则好像为了渲染而渲染，返回的不再是单纯的 self.response_body

循环渲染单个对象，还是渲染集合？

优先渲染一个集合(参数 `:collection`)，而不是一个局部模板或对象。

```
<% @advertisements.each do |ad| %>
  <%= render partial: "ad", locals: { ad: ad } %>
<% end %>

# 建议更改为

<%= render partial: "ad", collection: @advertisements %>
```

可以少敲几个字符，并且 Rails 对渲染集合做了一些优化，性能会变快一点。

另：渲染集合时，尽量不要省略 `:partial` 参数。

其它

Controller 里可以 render 完整的模板(template)、或局部模板(partial)，但 View 里不可指定渲染完整的模板(template).

页面里渲染 js.erb 有时候会有安全隐患，所以渲染后用 `escape_javascript` 处理。

`spacer_template` 在渲染之间穿插东西有时挺方便的，例如奇偶不同时我们设置颜色不同。

Translation

I18n 相关的 `t & translate` 和 `l & localize` 方法。

区别于 Active Model 的 Translation 模块。

~~Collector~~

我们在 `respond_to` 里可以响应的格式 Mime.

Collector 是 Abstact Controller 实现的。Action Controller 有自己的扩展，Active Mailer 也有自己的扩展。

对应：

```
# action 里面
respond_to do |format|
  format.html
  format.xml { render xml: @people }
end
```

调用 `format.html` 和 `format.xml` 等方法调用时会触发 `method_missing` ，之后由 `generate_method_for_mime` 元编程生成。

另外，如果请求没有指定格式，响应默认会按 format 里顺序执行，所以注意一下顺序。像一些搜索引擎就没有指定请求格式，所以一般的放 `format.html` 在前面。

目前 Rails 支持的 MIME(多用途互联网邮件扩展)，有：

> html text js css ics csv vcf
>
> png jpeg gif bmp tiff
>
> mpeg
>
> xml rss atom yaml
>
> multipart_form url_encoded_form
>
> json
>
> pdf zip

可以在 action_dispatch/http/mime_types.rb 里查看。

其它

Abstract Controller 部分模块在前面已经详细介绍了，下面对其余模块做简单描述，有利于对源码的阅读。

Asset Paths

以声明的形式，定义一些 assset 相关的类方法和实例方法。

```
config_accessor :asset_host, :assets_dir, :javascripts_dir,
                :stylesheets_dir, :default_asset_host_protocol,
:relative_url_root
```

Routes Helpers

引入 **Route** 相关的 **x_path** 和 **x_url** 辅助方法。

这里只是引入，这些方法在 RouteSet 里定义。

```
include Rails.application.routes.url_helpers
```

之后，就能调用和 Routing 相关的 helper 方法。

`routes.rb` 里定义的每一个路由规则都会有对应的 x_url 和 x_path 等 helper 方法可用，这里 include 了这些 helper 方法。

然后，Action Controller 和 Action Mailer 的 Railtie 又 extend Routes Helpers，所以可用。

~~Logger~~

提供 `logger` 方法打印日志。

~~Url For~~

简单封装了 ActionDispatch::Routing::UrlFor，然后给 Active Mailer 和 ActionController::UrlFor 使用。

也就是说，Action Controller 和 Action Mailer 在这里没有直接与 Action Dispatch 沟通。

Caching

perform_caching 配置是否启用缓存，cache_store 配置启用何种缓存方式。

页面相关服务端缓存

片段缓存相关，在这里可以单独拿出来讲。

页面相关的服务端缓存，由两个模块组成：Caching 和 Caching Fragments.

Caching 模块

- cache_store 有没有到位
- cache 相关 perform_caching 是否使用
- 视图缓存依赖

除以上外，它几乎什么也没做，定义了 `cache` 同名方法，保证了执行缓存需要的两个条件：`perform_caching = true` 并且 `cache_store` 是合法的。

```
def cache(key, options = {}, &block)
  if cache_configured?
    cache_store.fetch(ActiveSupport::Cache.expand_cache_key(key, :controller),
                      options, &block)
  else
    yield
  end
end
```

1) 想要关闭片段缓存，可以配置(开发环境下默认就是 false)：

```
config.action_controller.perform_caching = false
```

2) Caching stores

配置怎么存储缓存(默认使用 MemoryStore):

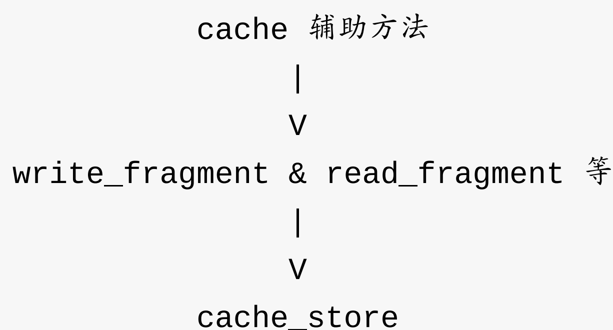
```
config.action_controller.cache_store = :memory_store
config.action_controller.cache_store = :file_store, '/path/to/cache/directory'
config.action_controller.cache_store = :mem_cache_store, 'localhost'
config.action_controller.cache_store = :mem_cache_store,
                                        Memcached::Rails.new('localhost:11211')
config.action_controller.cache_store = MyOwnStore.new('parameter')
```

3) view_cache_dependencies

Caching Fragments 模块

对片段缓存的一些操作。

这里的方法属于中间处理过程，封装对底层的操作(也就是 `cache_store`)，然后提供接口给我们上层使用(也就是 `cache` 辅助方法)。



提供了这么几个方法：

```
fragment_cache_key

fragment_exist?

read_fragment
write_fragment

expire_fragment
```

`expire_fragment` 你可以手动指定片段缓存过期的规则：

```
expire_fragment(controller: 'products', action: 'recent',
                action_suffix: 'all_products')
```

操作这几个方法的是 controller，而这个几方法操作的是 cache_store.

> Note: ActionView::Helpers::CacheHelper 里的 cache 方法用到了 read_fragment、write_fragment 和 fragment_cache_key.

默认片段缓存策略

视图包含两类内容：Controller 传递过来的实例变量(动态内容，必需手动设置)，本身的静态内容(自动设置)。 缓存要考虑这两种内容的更新。

还有就是视图里的显示是层层嵌套的，它们之间有时候是有关联的，这种情况也要考虑。

怎么生成的 Cache Key？默认策略是路径 + 动态内容 Cache Key + 静态内容 Cache Key：

```
views/projects/123-20120806214154/7a1156131a6928cb0026877f8b749ac9
      ^class   ^id ^updated_at    ^template tree digest
```

第一次访问(读缓存失败，写缓存)

```
Cache digest for
app/views/posts/show.html.erb: a357e54a8e1fdeff463f2da17cdc8197
Read fragment
views/posts/1-20140421062215459004000/a357e54a8e1fdeff463f2da17cdc8197
Write fragment
views/posts/1-20140421062215459004000/a357e54a8e1fdeff463f2da17cdc8197
```

· 第二次访问(读缓存成功)

```
Cache digest for
app/views/posts/show.html.erb: a357e54a8e1fdeff463f2da17cdc8197
Read fragment
views/posts/1-20140421062215459004000/a357e54a8e1fdeff463f2da17cdc8197
```

查看一下动态内容的 cache_key，方便对比

```
post.cache_key
# => "posts/1-20140421062215459004000"
```

更改静态内容(读缓存失败，写缓存；动态内容这部分 Cache Key 不变)

```
Cache digest for
app/views/posts/show.html.erb: 6e30019bd1127688840f7307cbe5cfbc
Read fragment
views/posts/1-20140421062215459004000/6e30019bd1127688840f7307cb
e5cfbc
Write fragment
views/posts/1-20140421062215459004000/6e30019bd1127688840f7307cb
e5cfbc
```

更改动态内容(在这里是update post. 读缓存失败，写缓存；静态内容这部分 Cache Key 不变)

```
Cache digest for
app/views/posts/show.html.erb: 6e30019bd1127688840f7307cbe5cfbc
Read fragment
views/posts/1-20140421064029939882000/6e30019bd1127688840f7307cb
e5cfbc
Write fragment
views/posts/1-20140421064029939882000/6e30019bd1127688840f7307cb
e5cfbc
```

如何才能完全手动管理缓存？(也许永远都没必要这么做)：

1. 不要使用动态内容做 key
2. 关闭静态内容加密

比如：

```
cache 'all_available_products', skip_digest: true
```

此时：

```
expire_fragment('all_available_products')
```

才会生效。

Action Controller

TODO

Metal - 加强的 Rack, 简陋的 Controller

加强的 Rack

意味着它符合 Rack 接口规范，可以直接使用它，创建出来的应用可以看做是一个 Rack application.

在 Rack 的基础上，它增加了 middleware_stack 的预处理。

和 Rack 一样，它的功能真的很有限。如你的项目做为 API 对外提供服务，不需要那么多功能，你可以尝试。

和 Rack 一样，相对来说它的性能比较高。如你的 Rails 项目对性能要求非常高，你可以尝试。

> Rails Metal is a subset of Rack middleware 可以参考【Rack - Ruby Web server 接口】章节了解更多关于 Rack 的内容。

简陋的 Controller

除了提供一个有效的 Rack 接口外，它几乎没有任何其它功能。

ActionController::Base 在它基础之上添加了多个类和模块，这使得功能得到增多，同时在性能上也会有相应损耗。如果你觉得这些功能不是必需的，或者性能的损耗是不可忍受的，你可以直接使用 Metal.

举个例子:

```
class HelloController < ActionController::Metal
  def index
    self.response_body = "Hello World!"
  end
end
```

在路由里添加相应代码，将请求转发到刚才的 HelloController#index 进行处理:

```
# config/routes.rb
get 'hello', to: HelloController.action(:index)
```

为了让 Route 能够很好转发，action 方法会返回一个有效的 Rack application.

主要做的事情

调用 middleware 进行预处理。

像 Rack application 一样小而完整的处理。

做出响应。

其它

一般模块名和同名目录都是有联系的，但 metal 不是，它单指的是 metal.rb 这个文件，和 metal/ 目录下的文件及内容没有关系。

- 直接使用 Metal 时，要清楚自己在做什么。

另外，要清楚的知道各个组件有什么用，添加是为了什么，去掉又会有什么影响。

- 为什么能够连续调用，原因：

你看每个 Rack Middleware 的 `call` 函数的最后一行，是不是都是 `@app.call(env)` ？这说明，它在调用下一个 middleware.

它们是一条封闭的链接，一直走下去，最后又会回到开头处，并且中间只要有一处断了，那整条链子就都走不通。

顺序是：默认是按 `use` 的顺序走下去，但 use 时你也是可以指定的。

> Note: @app 和 env 内容一直在变，但本质又一直没变。

不再推荐使用字符串引入 middleware 所以

```
middleware.use "Foo::Bar"
```

变为

```
middleware.use Foo::Bar
```

Metal 文件下的内容

Metal 是请求从路由到 Controller 的中转站。

类方法：

```
action

use
middleware & middleware_stack

controller_name
```

实例方法：

```
dispatch

content_type
content_type=

location
location=

params
params=

status
status=

performed?

response_body=

controller_name

env

url_for
```

`self.action` 和 `dispatch` 为转移到下一站场起到了很大的作用。

```
# Action Dispatch 转发过来的请求，要先经过层层的 middleware 处理，才能到达指定的 action.
def self.action(name, klass = ActionDispatch::Request)
  if middleware_stack.any?
    middleware_stack.build(name) do |env|
      new.dispatch(name, klass.new(env))
    end
  else
    lambda { |env| new.dispatch(name, klass.new(env)) }
  end
end
```

从堆栈里取 middleware 并处理。

`use` 方法把 middleware 放入 middleware stack，也很重要。

```
class PostsController < ApplicationController
  use AuthenticationMiddleware, except: [:index, :show]
end
```

除此之外，需要清楚：

```
class_attribute :middleware_stack
self.middleware_stack = ActionController::MiddlewareStack.new
```

Middleware Stack

继承于 Action Dispatch 的 MiddlewareStack，用于存放 middleware.
它的功能和父类一样，只是作用的层面不同。

Middleware 并不总是需要项目级别的，它也可以精确到某个 Controller，甚至是 action.

```
class PostsController < ApplicationController
  use AuthenticationMiddleware, except: [:index, :show]
end
```

`self.use` 是前面 Metal 提供的方法，可以添加 middleware 到堆栈，它们在最后执行。
(这里的堆栈指的可以是 ActionController::MiddlewareStack，也可以是 ActionDispatch::MiddlewareStack).

Metal 使用举例

原生的 Metal

在 Rails 里 metal 也属于 middleware，我们可以这么用：

```
# config/routes.rb
get 'hello' => 'hello#index'
# ...
```

```
# app/controllers/hello_controller.rb
class HelloController < ActionController::Metal
  def index
    self.response_body = "Hello World!"
  end
end
```

然后浏览器访问：http://localhost:3000/hello 可以获取刚才的内容。

日志会由原来的：

```
Started GET "/hello" for 127.0.0.1 at 2014-04-27 09:04:49 +0800
Processing by HelloController#index as HTML
Completed 200 OK in 1ms (ActiveRecord: 0.0ms)
```

变为：

```
Started GET "/hello" for 127.0.0.1 at 2014-04-27 08:57:07 +0800
```

加入 Rendering 模块

默认 ActionController::Metal 是没有提供渲染视图、模板和其它需要明确调用到 response_body=, content_type=, status= 等方法。如果你需要这些，可以引入它们：

```
class HelloController < ActionController::Metal
  include AbstractController::Rendering
  include ActionView::Layouts

  append_view_path "#{Rails.root}/app/views"

  def index
    render "hello/index"
  end
end
```

加入 Redirection 模块

想使用重定向相关代码，你也需要引入它们：

```
class HelloController < ActionController::Metal
  include ActionController::Redirecting
  include Rails.application.routes.url_helpers

  def index
    redirect_to root_url
  end
end
```

加入其它模块

参考 ActionController::Base 引入其它模块，以达到目的。

```
class ApiController < ActionController::Metal # 使用 Metal, 而不是
 Base
  include ActionController::Helpers
  include ActionController::Redirecting
  include ActionController::Rendering
  include ActionController::Renderers::All
  include ActionController::ConditionalGet

  # 需要响应 .json .xml 等不同格式的话，使用它。
  include ActionController::MimeResponds
  include ActionController::RequestForgeryProtection
  # 如果你需要 SSL
  include ActionController::ForceSSL
  include AbstractController::Callbacks
  # 想要知道 Controller 处理过程中，性能这方面的数据。
  include ActionController::Instrumentation
  # 需要转换 params 类型的话，可以使用它。
  include ActionController::ParamsWrapper
  include Devise::Controllers::Helpers

  # 在项目里使用路由相关的 helper 方法。
  include Rails.application.routes.url_helpers

  # 视图文件放在哪？
  append_view_path "#{Rails.root}/app/views"

  # 需要转换 params 类型的话，可以使用它。
  # { "person": { "name": "Kelby", "email": "leekelby@gmail.com"
 }}
  wrap_parameters format: [:json]

  # 根据客户端决定是否需要。(另，如果客户端不是浏览器的话，它会自动跳过)
  protect_from_forgery
end
```

上述代码，仅供参考。关于各模块及其作用，详情可以参考对应章节。

参考

Developing api with rails metal

API - Metal 的继承者

精简版的 ActionController::Base，为 API 而生。

它没有也不需要 layouts 和 templates rendering, cookies, sessions, flash, assets 等。

- 默认只有 ApplicationController 继承于它
- 没有默认渲染。所以需要明确指定渲染
- 除了渲染，也可以有重定向
- 可用 include 引入其它模块，比如 MimeResponds

它和 ActionController::Base 本质上是一样的，对比来说，它去掉了很多不必要的 Metal 增强组件，所以性能会比较快：

```
ActionController::API::MODULES
```

```
MODULES = [
  AbstractController::Rendering,

  UrlFor,
  Redirecting,
  ApiRendering,
  Renderers::All,
  ConditionalGet,
  BasicImplicitRender,
  StrongParameters,

  ForceSSL,
  DataStreaming,

  AbstractController::Callbacks,

  Rescue,

  Instrumentation,

  ParamsWrapper
]

MODULES.each do |mod|
  include mod
end
```

配置 api_only

路由资源默认生成几个 action

是否加载 View 相关的 dependencies rake

是否采用几个 middleware

generator 少生成几个目录

Base - Metal 的继承者

相对于 Metal，它包含了各个控制器上能够使用到的组件。所以使用它时，在性能上会比直接继承使用 Metal 慢得多，但我们可用的功能更丰富了。

我们看看 ActionController::Base 引入了哪些模块：

```
ActionController::Base::MODULES
```

```
MODULES = [
  AbstractController::Rendering,
  AbstractController::Translation,
  AbstractController::AssetPaths,

  Helpers,
  UrlFor,
  Redirecting,
  ActionView::Layouts,
  Rendering,
  Renderers::All,
  ConditionalGet,
  EtagWithTemplateDigest,
  RackDelegation,
  Caching,
  MimeResponds,
  ImplicitRender,
  StrongParameters,

  Cookies,
  Flash,
  RequestForgeryProtection,
  ForceSSL,
  Streaming,
  DataStreaming,
  HttpAuthentication::Basic::ControllerMethods,
  HttpAuthentication::Digest::ControllerMethods,
  HttpAuthentication::Token::ControllerMethods,
```

```
  AbstractController::Callbacks,

  Rescue,

  Instrumentation,

  ParamsWrapper
]

MODULES.each do |mod|
  include mod
end
```

引入了这么多模块，虽然方便了使用。但有的模块，我们用不到，浪费了。

如果对性能有很高要求，并且知道各个模块作用的话，可以适当去掉某些模块。

> 后文讲到的【API】是精简版的 Base，它也是直接继承于 Metal，在 API 模式里可用。

Metal 的增强模块

有一些模块和 Metal 一样也有 `process_action` 方法，并且它们也被 include 进了 Action Controller，并且我们自己定义的类没有重写这个方法。

根据 Ruby 的代码执行规则，执行 process_action 时它们都会被执行(并且这个方法执行顺序先于 Metal 同名的方法)。

源代码里，Metal 仅代表 metal.rb 这个文件，不包括与其同名的 metal 目录。

1) 继承 **Abstract Controller** 的财富

由 Metal 到 Base 继承而来。

2) 使用 **Action Dispatch** 的资源

主要是 Action Dispatch 的 http 和 middleware 下的内容。

包括但不限于：

处理 request、response 相关和 headers 相关。(Metal 也可以做，但这里得到了充分利用)

3) 协作 **Action View**

有很多类似的方法或丝丝关联，比如：渲染。

4) 少量的 **Active Model**

为了约定优于配置，很少的一部分方法和 Active Model 的 Naming 模块有关联。

包括但不限于：

```
wrap_parameters

polymorphic_url
polymorphic_path
```

5）提供很多方法，允许你在 **Controller** 或 **action** 里处理请求，并响应。

Redirecting

```
redirect_to

redirect_back
```

重要的部分就是可以根据不同的可选参数，计算出要重定向的 url.

使用举例：

```
# 字符串
redirect_to “www.rubyonrails.org”
redirect_to “/images/screenshot.jpg”
redirect_to articles_url

# :back
redirect_to :back

# Proc
redirect_to proc { edit_post_url(@post) }

# 其它可选参数，会用到 url_for 来处理

# record 对象
redirect_to post

# Hash
redirect_to action: “show”, id: 5
redirect_to({ action: 'atom' }, alert: "Something serious happen
ed")
```

相关、类似功能：

`url_for` 根据给定的参数和 default_url_options 和 routes.rb 里的路由定义，生成可用的 url.

`polymorphic_url` 根据传递的 record 对象，构建可用的 url.

极端情况下，才会发生：

```
  redirect_to
      |
      V
ActionController::UrlFor
      |
      V
   url_for
      |
      V
AbstractController::UrlFor
      |
      V
ActionDispatch::Routing::UrlFor
      |
      V
ActionDispatch::Routing::PolymorphicRoutes
      |
      V
polymorphic_url
      |
      V
   ... ...
```

`redirect_back` 相当于来的 `redirect_to :back` 但它可接受 `:fallback_location` 参数，以应对 Referer 不存在的问题。

Head

返回一个内容为空的 **response.**

有时候(作为 Web service 时)，响应内容可能只需要一个状态码，其它内容都不需要。

直接用 `head` 方法:

```
head :created, :location => person_url(@person)
```

注意：使用 head 只是设置了 respond_body，如果程序还没有结束，那么 Controller#action 层面的后续代码还是会执行的。

结果和使用 render nothing: true 类似，所以你也可以这么做：

```
headers['Location'] = person_url(@person)
render :nothing => true, :status => "201 Created"
```

当然，如果有多个地方使用到此功能的话，render 写起来还是挺让人头疼的。

附：对照表

Response Class	HTTP Status Code	Symbol
消息	100	:continue
	101	:switching_protocols
	102	:processing
成功	200	:o
	201	:created
	202	:accepted
	203	:non_authoritative_information
	204	:no_content
	205	:reset_content
	206	:partial_content

	207	:multi_status
	208	:already_reported
	226	:im_used
重定向	300	:multiple_choice
	301	:moved_permanently
	302	:found
	303	:see_other
	304	:not_modified
	305	:use_proxy
	306	:reserved
	307	:temporary_redirect
	308	:permanent_redirect
客户端错误	400	:bad_reques
	401	:unauthorized
	402	:payment_required
	403	:forbidden
	404	:not_found
	405	:method_not_allowed
	406	:not_acceptable
	407	:proxy_authentication_required
	408	:request_timeout
	409	:conflict
	410	:gone
	411	:length_required
	412	:precondition_failed
	413	:request_entity_too_large
	414	:request_uri_too_long
	415	:unsupported_media_type
	416	:requested_range_not_satisfiable
	417	:expectation_failed

	422	:unprocessable_entity
	423	:locked
	424	:failed_dependency
	426	:upgrade_required
	428	:precondition_required
	429	:too_many_requests
	431	:request_header_fields_too_large
服务端错误	500	:internal_server_erro
	501	:not_implemented
	502	:bad_gateway
	503	:service_unavailable
	504	:gateway_timeout
	505	:http_version_not_supported
	506	:variant_also_negotiates
	507	:insufficient_storage
	508	:loop_detected
	510	:not_extended
	511	:network_authentication_required

Conditional Get - HTTP Cache

页面相关的客户端缓存

根据 ETag 和 Last-Modified 来决定是否渲染页面，可充分利用客户端(例如浏览器)的缓存，在 Rails 里属于 Controller#action 层面的缓存。

它和 Rails.cache 等缓存机制都不一样，它用的是状态头和浏览器的特性，并不属于应用层面的缓存。

类方法：

```
etag
```

`etag` 把缓存元素加入到 etaggers 里。之后 cache 相关的方法会调用到 etaggers(即调用到这些缓存元素)，进而通过它们影响 HTTP 缓存结果。
这里添加的元素对当前 Controller 下所有的 action 都会起作用。

实例方法：

```
fresh_when

# 还有
expires_in
expires_now

stale?

http_cache_forever
```

> Rails 5 新增 100 年才过期的 http_cache_forever

`fresh_when` 根据传入的参数，设置 response 里和 HTTP 缓存有关的字段，并完成响应。只对当前 action 起作用，用到了上面提到的 etaggers，影响 ETag 和 last_modify 的值。

`stale?` 不只是单纯的询问，它用到了 fresh_when，所以除了返回 boolean 外，还会影响 response.

`fresh_when` 和 `stale?` 都可以传递 `:template` 参数以便指定模板。(这部分由【Etag With Template Digest】进行处理)

`expires_in` 设置 response 里和 HTTP 缓存有关的字段。
`expires_now` 设置 response 里和 HTTP 缓存有关的字段。

注意：HTTP 缓存有关的字段有多个，它们设置的是不同字段，或同字段但不同值。

使用举例：

```
# 默认传递 @post 给 posts/show 模板进行求值，我们希望改变这点，传递 @post
 给 widges/show 求值
fresh_when @post, template: 'widgets/show'

# 只对 @post 求值，不带入模板内容
fresh_when @post, template: false
```

支持设置 HTTP header 里的 etag 和 last modified 就意味着当所请求的资源没有更改过时，Rails 可以返回一个的响应。这可以节省你的带宽资源和时间。

> Note: 一般的，etag 相关设置只是节省了中间数据传输的网络资源，但在服务器上的计算并没有减少。

Rails 5 可传递集合给 fresh_when 或 stale?
例如：

```
# 之前
def index
  @articles = Article.all
  fresh_when(etag: @articles, last_modified: @articles.maximum(:
updated_at))
end

# 之后
def index
  @articles = Article.all
  fresh_when(@articles)
end
```

Conditional Get 其它

相关概念

1) ETag 过程如下：

- 客户端请求一个页面 A，服务器返回页面 A，并在给 A 加上一个 ETag.
- 客户端展现该页面，并将页面连同 ETag 一起缓存。
- 客户再次请求页面 A，并将上次请求时服务器返回的 ETag 一起传递给服务器。
- 服务器检查该 ETag，并判断出该页面自上次客户端请求之后还未被修改，直接返回响应 304（未修改——Not Modified）和一个空的 response body.

2) Last-Modified 过程如下：

在浏览器第一次请求某一个 URL 时，服务器端的返回状态会是 200，内容是客户端请求的资源，同时有一个 Last-Modified 的属性标记此文件在服务期端最后被修改的时间，格式类似这样：

```
Last-Modified : Fri , 12 May 2006 18:53:33 GMT
```

客户端第二次请求此 URL 时，根据 HTTP 协议的规定，浏览器会向服务器传送 If-Modified-Since 报头，询问该时间之后文件是否有被修改过：

```
If-Modified-Since : Fri , 12 May 2006 18:53:33 GMT
```

如果服务器端的资源没有变化，则自动返回 HTTP 304 (Not Changed) 状态码，内容为空，这样就节省了传输数据量。当服务器端代码发生改变或者重启服务器时，则重新发出资源，返回和第一次请求时类似。从而保证不向客户端重复发出资源，也保证当服务器有变化时，客户端能够得到最新的资源。

3) 一个典型的 Response headers:

```
Last-Modified: Sun, 09 Nov 2014 05:25:32 GMT
Etag: "4808a7a249c9fd981be8ba390f55ce8a"

Cache-Control: max-age=0, private, must-revalidate
Set-Cookie: _sample_app_session=some-thing%3D%3D--some-thing-els
e; path=/; HttpOnly
X-Request-Id: 02d61994-90aa-41e4-be96-b65b68914c13
X-Runtime: 0.005787
Server: thin 1.6.2 codename Doc Brown
Date: Sun, 09 Nov 2014 05:42:53 GMT
X-Frame-Options: SAMEORIGIN
X-Xss-Protection: 1; mode=block
X-Content-Type-Options: nosniff
Content-Type: text/html; charset=utf-8
Content-Length: 665
```

和所有缓存一样，怎么生成 cache_key ？

最新，什么也没改。

```
f9d7568ac634fc9ad270f2348d5f3b41
```

更改表态内容：

```
d9cc40e9e225a3e738c68b01f17ef4b0
update_columns 7128cbb34d84aa096ee62d69d1599a32
update_attributes 98eac754b15c61f4c8b4f79b7f0a5645
```

> Note: 默认动态或静态内容有变，ETag 都会更新；都不改变，才不更新。如果手动设置了 fresh_when 反而会获取到旧数据。

如果 fresh_when 匹配，将直接返回结果给最终用户。但后面的代码并未退出，还会执行，只到遇到结束标识为止。也就是说 Controller#action 后续有代码的话，则还会执行，但 View 里的代码不会执行了

这里区别于响应：

```
# 之前，还需要渲染视图
Completed 200 OK in 84ms (Views: 10.7ms | ActiveRecord: 2.7ms)

# 之后，不需要渲染视图
Completed 304 Not Modified in 3ms (ActiveRecord: 0.1ms)
```

相关代码

`fresh_when` 方法：

```
def fresh_when(record_or_options, additional_options = {})
  # ... 参数处理，略

  # 设置 response 部分属性
  response.etag = combine_etags(options) if options[:etag] || op
tions[:template]
  response.last_modified = options[:last_modified] if options[:l
ast_modified]
  response.cache_control[:public] = true           if options[:p
ublic]

  # 调用 head 方法，返回状态码 304
  head :not_modified if request.fresh?(response)
end
```

可选参数 etag、template、last_modified 和 public.

`head` 方法：

```
def head(status, options = {})
  # ... 设置 status、location、content_type 等，略

  if include_content?(self._status_code)
    self.content_type = content_type || (Mime[formats.first] if
formats)
    self.response.charset = false if self.response
    self.response_body = " "
  else
    headers.delete('Content-Type')
    headers.delete('Content-Length')
    self.response_body = ""
  end
end
```

`request.fresh?` 方法：

```
def fresh?(response)
  last_modified = if_modified_since
  etag          = if_none_match

  return false unless last_modified || etag

  success = true
  success &&= not_modified?(response.last_modified) if last_modi
fied
  success &&= etag_matches?(response.etag) if etag
  success
end
```

其它

fresh_when 可以通过工具(如：Chrome 插件"Advanced REST client")查看 ETag，检测是否起作用以及是否正确。

> Note: ETag 和已经被废除 cache_page 的区别，前者是 Web 服务器级别，后者是应用服务器级别。如果页面没有更改，ETag 返回的是 "304 Not Modified"，其他什么都不用干，连网络带宽都省了。cache_page 还要读取 public/ 目录下的静态 HTML 文件，减少了计算过程，但还是需要网络传输页面内容。

Etag With Template Digest

用 **Digestor** 给当前静态模板(对应 **controller_name/action_name)**加密，使之构成 **Etag** 的一部分。

前面提到的 `fresh_when` 匹配的只是动态内容。静态内容怎么生成 cache_key，怎么过期？(比如开发环境下，我们会经常修改静态内容)

答案是：给静态内容做 md5 加密。每一次更改静态内容，对应的 Hash 值都会改变。在开发环境下，每次更新都会刷新页面；在生产环境下，重启服务器后会刷新。

默认该特性已启用，也就是说 Action View 静态内容已经加密(区别于片段缓存里对静态内容的加密)。

可以设置取消：

```
config.action_controller.etag_with_template_digest = false
```

取消后，静态内容不经过加密，更改它们，Hash 值并不会更改。就会导致生产环境下，即使重启服务器得到的还是原来的内容。

本设置影响的是 ETag，缓存的是当前用户浏览的内容，他人或使用新浏览器访问页面，不受影响。

用到了 ConditionalGet::etag 和 Digestor::digest，注意它只用于条件判断，本身不会更改内容。

3 个级别的验证，分别是 Basic、Digest 和 Token.

设置头部消息 "WWW-Authenticate"，获取 `headers["WWW-Authenticate"]`

我们最最常用的是：

```
authenticate_or_request_with_http_basic(realm = "Application", &
login_procedure)
```

Basic

对应头部字段及内容：

```
headers["WWW-Authenticate"] = %(Basic realm="#{realm.gsub(/"/, "")}")
```

类方法：

```
http_basic_authenticate_with
```

使用举例：

```
class PostsController < ApplicationController
  http_basic_authenticate_with name: "dhh", password: "secret", except: :index

  def index
    render plain: "Everyone can see me!"
  end

  def edit
    render plain: "I'm only accessible if you know the password"
  end
end
```

`http_basic_authenticate_with` 除 :name 和 :password 选项外，一般还可设置 :realm 做为提示信息。它已经封装了 authenticate_or_request_with_http_basic 方法。

`http_basic_authenticate_with` 最常用的验证方式。

Controller 方法：

```
authenticate_with_http_basic
request_http_basic_authentication

authenticate_or_request_with_http_basic
```

`authenticate_or_request_with_http_basic` 简单的封装了其余两个方法。

使用举例：

```
class ApplicationController < ActionController::Base
  before_action :set_account, :authenticate

  protected
    def set_account
      @account = Account.find_by(url_name: request.subdomains.fi
rst)
    end

    def authenticate
      case request.format
      when Mime::XML, Mime::ATOM
        # 使用验证
        if user = authenticate_with_http_basic do |u, p|
            @account.users.authenticate(u, p) # <- 这里
          end

          @current_user = user
        else
          # 使用验证
          request_http_basic_authentication # <- 这里
        end
      else
        if session_authenticated?
          @current_user = @account.users.find(session[:authentic
ated][:user_id])
        else
          redirect_to(login_url) and return false
        end
      end
    end
end
```

其它方法：

```
auth_param
auth_scheme

authenticate

authentication_request

decode_credentials
encode_credentials

user_name_and_password

has_basic_credentials?
```

使用举例：

```
def test_access_granted_from_xml
  get(
    "/notes/1.xml", nil,
    'HTTP_AUTHORIZATION' =>
     ActionController::HttpAuthentication::Basic.encode_credenti
als(
       users(:dhh).name,
       users(:dhh).password
     )
  )

  assert_equal 200, status
end
```

Digest

对应头部字段及内容：

```
headers["WWW-Authenticate"] = %(Digest realm="#{realm}", qop="au
th", algorithm=MD5,
                                nonce="#{nonce}", opaque="#{opaq
ue}")
```

Controller 方法：

提供方法：

```
authenticate_with_http_digest
request_http_digest_authentication

authenticate_or_request_with_http_digest
```

使用举例：

```
require 'digest/md5'
class PostsController < ApplicationController
  REALM = "SuperSecret"
  USERS = {"dhh" => "secret",
           # plain text password
           # ha1 digest password
           "dap" => Digest::MD5.hexdigest(["dap", REALM, "secret"
].join(":"))}

  before_action :authenticate, except: :index

  def index
    render plain: "Everyone can see me!"
  end

  def edit
    render plain: "I'm only accessible if you know the password"
  end

  private
    def authenticate
      # 使用验证
      authenticate_or_request_with_http_digest(REALM) do |userna
me| # <- 这里
        USERS[username]
      end
    end
end
```

`authenticate_or_request_with_http_digest` 简单的封装了其余两个方法。从名字可知，如果提供的是普通文本则直接接受；如果提供的是 md5 加密，则先(自动)解密再接受。

其它方法：

```
authenticate
authentication_header
authentication_request

decode_credentials
decode_credentials_header
encode_credentials

expected_response
ha1
nonce
opaque
secret_token

validate_digest_response
validate_nonce
```

Token

对应头部字段及内容：

```
headers["WWW-Authenticate"] = %(Token realm="#{realm.gsub(/"/, ""
)}")
```

Controller 方法：

提供方法：

```
authenticate_with_http_token
request_http_token_authentication

authenticate_or_request_with_http_token
```

`authenticate_or_request_with_http_token` 简单的封装了其余两个方法。

使用举例：

```
class PostsController < ApplicationController
  TOKEN = "secret"

  before_action :authenticate, except: :index

  def index
    render plain: "Everyone can see me!"
  end

  def edit
    render plain: "I'm only accessible if you know the password"
  end

  private
    def authenticate
      # 使用验证
      authenticate_or_request_with_http_token do |token, options
|
        token == TOKEN
      end
    end
end
```

再次举例：

```
class ApplicationController < ActionController::Base
  before_action :set_account, :authenticate

  protected
    def set_account
      @account = Account.find_by(url_name: request.subdomains.fi
rst)
    end

    def authenticate
      case request.format
      when Mime::XML, Mime::ATOM
        # 使用验证
        if user = authenticate_with_http_token do |t, o|
          @account.users.authenticate(t, o)
          end

          @current_user = user
        else
          # 使用验证
          request_http_token_authentication
        end
      else
        if session_authenticated?
          @current_user = @account.users.find(session[:authentic
ated][:user_id])
        else
          redirect_to(login_url) and return false
        end
      end
    end
end
```

其它方法：

```
authenticate
authentication_request

encode_credentials

params_array_from

rewrite_param_values

token_and_options

raw_params
token_params_from
```

传递 token 时，可以传递额外的数据，如：

```
Authorization: Token token="abc", nonce="def"
```

然后使用 `token_and_options` 获取这些数据。

使用举例：

```
def test_access_granted_from_xml
  get(
    "/notes/1.xml", nil,
    'HTTP_AUTHORIZATION' =>
     ActionController::HttpAuthentication::Token.encode_credenti
als(users(:dhh).token)
  )

  assert_equal 200, status
end
```

Token 验证的部分特点：

1. 不能直接明文出现在 url 里。
2. 通过 curl -H 'Authorization: Token token="x"' 传递数据。
3. 通常用于制作程序接口。

4. 不提供弹窗输入，其余两种验证方式提供。

Streaming

改变渲染顺序。

Rails 默认的渲染过程：先是模板，然后是数据库查询，最后才是布局。

```
HTTP request -> dynamic content generation -> static head genera
tion -> HTTP response
```

具体一点：它先执行 `yield` 里的内容，渲染模板，最后才是加载 assets/layouts.

Streaming 可以改变一下顺序，按布局来渲染。布局先显示，对于用户体验可能更好一点，还有就是这会使得 JS 和 CSS 的加载顺序比平时提前。 变成(这里只是类似)：

```
HTTP request -> static head generation -> HTTP response
             -> dynamic content generation -> HTTP response
```

头部消息加上了：

```
Transfer-Encoding: chunked
```

如何使用？

Streaming 没有对外提供方法，在 `render` 的时候，加上 `:stream` 参数即可：

```
class PostsController
  def index
    @posts = Post.all
    render stream: true # 仅作用于模板，json、xml 等格式的数据不行
  end
end
```

先返回 HTML 的 head 部分，之后返回 body 部分；而不是像之前全部渲染完全才一起返回 head 和 body.
一般我们都把 js 和 css 放 head 里，这意味着它们会先于页面内容返回。

注意事项：

注意：使用 Streaming 特性后后，HTML 里的 head 部分会先返回，不受页面内容的影响。 如果原来的 head 部分依赖于页面，需要做对应修改。

举例：如果 `title` 是根据页面动态生成的，使用 Streaming 后就会出现缺失的情况，需要更改原有代码。可以改成使用 `provide` 和 `yield` ：

```
# projects/index.html.erb
<% provide :title, "Projects" %>
```

```
# layouts/application.html.erb
<title><%= yield :title %></title>
```

再举例：使用此特性后，将不能动态生成 `cookie` 和 `session` 内容。

再举例：并不是所有应用服务器(例如默认的 Webrick)都支持此特性。

使用之前，请搞清楚它要解决的问题，以及带来的新问题。还有，不要和【Live】里的 `stream` 混淆了。

> Note: 可通过命令 curl -i localhost:3000 查看使用效果。

Live

实时推送消息。可用于构建实时聊天之类等。

尽量不要和 Streaming 搅在一起，它们有类似之处，但不要混淆了。

默认不使用 SSE

默认情况下不使用 SSE，而是 Buffer，举例：

```
class MyController < ActionController::Base
  # 步骤 1
  include ActionController::Live

  def stream
    # 步骤 2
    response.headers['Content-Type'] = 'text/event-stream'

    100.times {
      # 步骤 3 直接使用 response.stream
      response.stream.write "hello world\n"
      sleep 1
    }
  ensure
    # 步骤 4
    response.stream.close
  end
end
```

建议 web server 使用 gem 'puma'，开发环境下默认使用的 WEBrick 不支持。Unicorn(响应比较快，并且对超时比较严格)、Rainbows! 或 Thin 也可以。

使用 SSE

SSE 全称 Server Sent Event，HTML5 服务器发送事件。

提供方法：

```
close

write
```

使用举例：

```
class MyController < ActionController::Base
  # 步骤 1
  include ActionController::Live

  def index
    # 步骤 2
    response.headers['Content-Type'] = 'text/event-stream'

    # 步骤 3 sse 封装了 response.stream
    sse = SSE.new(response.stream, retry: 300, event: "event-nam
e")

    # 步骤 4
    sse.write({ name: 'John'})
    sse.write({ name: 'John'}, id: 10)
    sse.write({ name: 'John'}, id: 10, event: "other-event")
    sse.write({ name: 'John'}, id: 10, event: "other-event", ret
ry: 500)
  ensure
    # 步骤 5
    sse.close
  end
end
```

实例方法

```
process

set_response!

response_body=

log_error
```

`process` 常见的同名方法，这里主要是另开一线程进行处理请求。

`set_response!` 也是常见的同名方法，这里主要是设置 response 为 Live::Response 的实例对象。

`response_body=` 主要是用于确保 response 用完后关闭。

`log_error` 记录错误(有的话)。

注意事项

- `content_for` 根据情况，有的要改为 `provide`
- 如果模板里有更改 Headers, cookies, session and flash 的值，将不起作用
- 部分 middleware 将不能再使用，如：Rack::Bug、Rack::Cache
- 异常报告和 Web server 的支持

参考

#401 ActionController::Live
Is it live?

Mime Responds & Collector

实例方法

```
respond_to
```

`respond_to(*mimes, &block)` - 全部内容(包含类型和内容)

respond_to 可以指定 format, 如 html. 而在 format 里又可以指定 variant, 如：phone.

```
format.html.phone # 行内风格

# 或

format.html{ |variant| variant.phone } # 代码块风格
```

Collector

扩展了 AbstractController::Collector，并且，增加了对"变种"的支持。

常用代码：

```
respond_to do |format|
  # format.class => ActionController::MimeResponds::Collector

  format.html
  format.js
end
```

可见，respond_to 里的 `format` 是其实例对象。

> Note: 想知道这里的 `format.html` 和 `format.js` 等方法是如何生效的，可以参考 Abstract Controller 的【Collector】章节。

提供方法：

```
all & any

custom

negotiate_format

response
```

此外，`format.html` 等对应着 Collector，而变种 `format.html.phone` 等对应着 Variant Collector.

有时候，如果遇到出错 ActionController::UnknownFormat，可考虑响应所有格式：

```
respond_to do |format|
  format.html { ... }
  format.all  {render :text => "Only HTML supported"}
end
```

Renderers 增删渲染器

针对 respond_to 里面不同的 format 会有不同的渲染器对它们进行处理。

实例方法：

```
render_to_body
```

`render_to_body` 在其它地方有同名方法，执行渲染时就会触发它，这里就隐藏着魔法。

还有实例方法 _render_to_body_with_renderer，在实现过程中很重要。

类方法：

```
add

remove
```

1) 使用 add 添加渲染器：

```
ActionController::Renderers.add :csv do |obj, options|
  filename = options[:filename] || 'data'
  str = obj.respond_to?(:to_csv) ? obj.to_csv : obj.to_s

  send_data str, type: Mime::CSV,
                 disposition: "attachment; filename=#{filename}.
csv"
end
```

2) 之后要注册：

```
Mime::Type.register "application/csv", :csv
```

(在后面的 Mime Type register 章节还会提及)

3) 使用刚才添加的渲染器：

```
def show
  @csvable = Csvable.find(params[:id])

  respond_to do |format|
    format.html
    # 对应这里的 format.csv
    format.csv { render csv: @csvable, filename: @csvable.name }
  end
end
```

移除渲染器：

```
ActionController::Renderers.remove(:csv)
```

其它类方法：

```
use_renderer & use_renderers
```

默认 Action Controller 带着的渲染器:

```
ActionController::Renderers::RENDERERS
=> #<Set: {:json, :js, :xml}>
```

Params Wrapper

对请求过来的 **params** 进行预处理。通过 API 发送请求的话，使用它，特别方便。

类方法：

```
wrap_parameters
```

比如，实际发送的是：

```
{"name": "Konata"}
```

预处理过后，可以变成：

```
{"user": {"name": "Konata"}}
```

或变成：

```
{"name" => "Konata", "user" => {"name" => "Konata"}}
```

或变成其它样子，数据类型也可以改变。

开放 API 时，此特性用得最多，此时请求格式一般为 'application/json'

使用举例：

```
# 默认
wrap_parameters false

# 原 params 复制一份，使用 :person 为 root 元素
wrap_parameters :person

# 原 params 复制一份，使用 :person 为 root 元素
wrap_parameters Person

# 原 params 复制一份，转化成 XML 格式，使用默认的 root 元素
wrap_parameters format: :xml

# 原 params 复制一份，但只要 :username 和 :title 部分，使用默认的 root
 元素
wrap_parameters include: [:username, :title]

# 原 params 复制一份，但排除 :title 部分，使用默认的 root 元素
wrap_parameters :exclude => :title
```

`include_root_in_json` 一个功能上恰好和它有类似之处的方法(一个在 Active Model，另一个在 Action Controller)。

```
Post.to_json

# 默认，没有 root 元素
{title: 'hello world'}

# 如果 include_root_in_json = true
{"post": {title: 'hello world'}}
```

Request Forgery Protection

防止 - 跨站请求伪造。

请求伪造保护。

什么伪造？

跨站请求伪造(CSRF)。

怎么保护？

生成字符串，放 session 里，请求过来时要校验。

类方法：

```
protect_from_forgery
```

方法里调用了：

```
before_action :verify_authenticity_token
```

等内容，使用举例：

```
class ApplicationController < ActionController::Base
  protect_from_forgery

  # 跳过 protect_from_forgery
  skip_before_action :verify_authenticity_token, if: :json_reque
st?

  protected

  def json_request?
    request.format.json?
  end
end
```

`protect_from_forgery` 可选参数 :only、:except、:with 和 :prepend

当请求未证实，可选择怎么处理。可选：

```
:exception     # 抛 ActionController::InvalidAuthenticityToken 异常。
:reset_session # 重置 session.
:null_session  # 使用空的 session 代替，但不删除(这是默认选择，可用 :with 指定)
```

实例方法：

```
verify_authenticity_token
```

```
handle_unverified_request
```

当请求未证实，会使用 `handle_unverified_request` 来处理。 对应上面的 Exception、Reset Session、Null Session.

由两部分组成：Parameters 和 Strong Parameters.

Parameters

Parameters 继承于 Hash With Indifferent Access，而 Hash With Indifferent Access 又继承于 Hash. 所以它的实例对象类似于 Hash 的实例对象。

提供 `params` 这个对象可以使用的方法(部分与 Hash 实例方法类似)：

```
permit

require & required

extract!

permit!

select!

permitted?

permitted=

[]
```

```
delete, dup

each & each_pair

fetch

to_h

slice

transform_values

converted_arrays
```

我们可以在 Rails 之外创建自己的 params 对象：

```
require 'action_controller/parameters'

params = ActionController::Parameters.new({
  person: {
    name: 'Francesco',
    age:  22,
    role: 'admin'
  }
})

# require(key) 和 permit(*filters) 方法
permitted = params.require(:person).permit(:name, :age)
permitted             # => {"name"=>"Francesco", "age"=>22}
permitted.class       # => ActionController::Parameters
# permitted? 方法
permitted.permitted? # => true

Person.first.update!(permitted)
# => #<Person id: 1, name: "Francesco", age: 22, role: "user">
```

可以直接使用 `permit!` 允许更新指定参数：

```
@user.update_attributes(params[:user].permit!)
```

配置默认的 permitted parameters

```
config.always_permitted_parameters = %w( controller action forma
t )
```

attributes.permitted? 与 ForbiddenAttributesProtection

因 Base 与 API 都有 `include StrongParameters` 并且仅提供 `params` 和 `params=` 方法，所以有理由相信在 Controller 和 View 里通过 params 给 record 对象属性赋值(AttributeAssignment)的话，都会询问一遍是否 `permitted?` 如果包含未被允许更新的字段，会抛 ForbiddenAttributesError 错误。

Strong Parameters

我们在 Controller 里常用的 `params` 就是这里提供的。

`params` 实际上是 Parameters 实例对象，我们可以对它的属性进行读、写操作。

```
params

params=
```

另外，要说明：

```
params == request.parameters
# => true
```

并不表明它和 `request.parameters` 是完全等价的，后者是 ActiveSupport::HashWithIndifferentAccess 实例对象。

这个对象的值是什么？- 表单数据或传递过来的，加上 :controller 和 :action

```
Processing by PostsController#create as HTML
Parameters: {"utf8"=>"✓",
             "authenticity_token"=>"kJttlgy9ptyuFS5TXrE95HFwKdhf
7p74yuFZl73Lvxg=",
             "post"=>{"title"=>"hello world"},
             "commit"=>"Create Post"}

params
=> {"utf8"=>"✓",
    "authenticity_token"=>"kJttlgy9ptyuFS5TXrE95HFwKdhf7p74yuFZl
73Lvxg=",
    "post"=>{"title"=>"hello world"},
    "commit"=>"Create Post",
    "action"=>"create",
    "controller"=>"posts"}
```

Data Streaming

```
send_data
send_file
```

`send_data` 类似 render :text
`send_file` 类似 response_body = file

它们发送方式类似，并且实现上也有相同的 `send_file_headers!`

但 `send_data` 可以发送的是数据，可以是非文件，也就是说即使是字符串也可发送。(比如动态生成的内容)

而 `send_file` 只能先有文件，才能发送。

相关可选参数：:filename、:type、:disposition、:status 或 :url_based_filename 可以查看 API 文档了解。

一般的，动态生成或一次性的内容，使用 send_data 比较好；纯文件或内容可供多次下载的，使用 send_file 比较好。

另，实际项目里，静态资源还可以通过 Web 服务器(Nginx/Apache)发送，应用只要提供 URL 即可。此时，仍然可以和原来一样调用 send_file 方法，但真正返回数据的时候，Web服务器会自动忽略掉应用服务器的 response.

```
# config/environments/production.rb

config.action_dispatch.x_sendfile_header = "X-Sendfile" # for Apache
config.action_dispatch.x_sendfile_header = 'X-Accel-Redirect' # for NGINX
```

Force SSL

实例方法：

```
force_ssl_redirect
```

重定向到 https 链接, url 会改变。
重定向这部分用到了 redirect_to 方法，提供可选参数 :status, :flash, :alert, :notice

如果没有(用参数)指定链接，则用当前链接，但会以 https 协议访问。可用于注册、登录等页面。 链接这部分用到了 url_for 方法，提供可选参数 :protocol, :host, :domain, :subdomain, :port, :path

上面是实例方法，对单个 action 有用；
下面这是类方法，它封装了 force_ssl_redirect，对整个 Controller 有用。

类方法：

```
force_ssl
```

可指定 action 跳转，可选参数 :only, :except；或根据条件跳转，可选参数 :if, :unless

使用举例：

```
class AccountsController < ApplicationController
  force_ssl if: :ssl_configured?

  def ssl_configured?
    !Rails.env.development?
  end
end
```

Flash

Action Controller 和 Action Dispatch 都有 Flash 相关的代码。

基本使用

类似 Hash，设置 flash

```
class PostsController < ActionController::Base
  def create
    # save post
    flash[:notice] = "Post successfully created"
    redirect_to @post
  end

  def show
    # ...
  end
end
```

类似 Hash, 读取 flash

```
# show.html.erb
<% if flash[:notice].present? %>
  <div class="notice"><%= flash[:notice] %></div>
<% end %>
```

alert 和 notice

因为 alert 和 notice 类型的 flash 太常见，所以提供了语法糖，你还可以这么写：

```
# 设置
flash.alert = "You must be logged in"
flash.notice = "Post successfully created"

# 读取
flash.alert
flash.notice
```

> Note: 其它 flash_type 默认不能这么写

上面是由 Action Dispatch 提供，下面由 Action Controller 提供。

add_flash_types 方法

觉得上面的写法还是不够简短，觉得 notice 和 alert 类型不够用？使用 `add_flash_types`

```
# app/cotrollars/pplication_controller.rb
class ApplicationController < ActionController::Base
  add_flash_types :warning, :success, :danger
end

# View 代码
<%= warning %>

# Controller 代码
redirect_to user_path(@user), warning: "Incomplete profile"
```

两种效果：视图里可以直接有同名 `warning` 辅助方法，`redirect_to` 里可直接使用 `:warning`.

它们和 flash[:warning] 或 flash.warning 和 flash: { warning: "Incomplete profile" } 效果一样。

还是 alert 和 notice

alert 和 notice 默认已经使用 add_flash_types

经验：默认 Rails 提供了上述类型的 flash，实际情况中一般是不够用的（至少对应红、黄、绿）3种级别的消息。所以，建议您用同样的方法添加自己的 flash.

flash.now[:flash_type]

也许，你还看过一种写法 `flash.now[:flash_type]`

flash[:flash_type] 消息的生命周期可到下一个 action. 所以，通常搭配 redirect_to 使用。
flash.now[:flash_type] 消息的生命周期仅限于本 action. 所以，通常搭配 render 使用。

以 update 为例：如果成功则跳转到新页面，那么可用 flash[:flash_type]; 失败则停留在当前页面，那么可用 flash.now[:flash_type].

另，当你意外的发现提醒消息在一个页面出现，在下一个页面还是出现，不妨改为 flash.now[:flash_type] 试试。

Helpers

```
helpers
```

本质是 ActionView::Base 的实例对象：

```
ApplicationController.helpers.class
# => ActionView::Base
```

rails 控制台里的 `helper` 方法指的就是它。

除上述方法外，还有：

```
all_helpers_from_path

helper_attr

modules_for_helpers
```

`helper_attr` 用到了 AbstractController::Helpers 里的 helper_method 方法。

另外，有配置：

```
# 默认为 true
config.action_controller.include_all_helpers = false
```

可以设置单个 Controller 只加载和它名字对应的 Helper 模块，而不是所有的 Helper(所有的 helper 方法)。

除上述描述外，因为它 `include AbstractController::Helpers` 所以它还有两个比较重要的方法可用。

Cookies

提供了 `cookies` 方法，本质是 request.cookie_jar

~~Implicit Render~~

默认渲染模板。

```
default_render

method_for_action
```

我们在 Controller#action 里，没有 render 模板或返回数据时 ... 就会用到它。

~~Instrumentation~~

当执行以下同名方法时，会发送消息，及有时间记录(以观察性能等指标)。

```
process_action

redirect_to

render

send_data
send_file
```

用到了 `ActiveSupport::Notifications.instrument` 和 `Benchmark.ms`

~~Rendering~~

对一般的 render_to_body 和 render_to_string 稍微做处理。

~~Rescue~~

执行到具体 action 抛异常时会检测是否需要抛异常，如果是的话，抛异常。

有 `process_action` 同名方法。

`show_detailed_exceptions?` (默认是 false，但本地请求的话是 true)，可配置 config.consider_all_requests_local

`rescue_with_handler` (封装了 ActiveSupport::Rescuable 的同名方法)

~~Url For~~

`url_for` 方法的组成部分。

~~Basic Implicit Render~~

用于 API 模式时，如果没有渲染或重定向，则返回 204（no_content）。

其它

Metal 增强模块和 Middleware 的区别

Middleware 是 Action Dispatch 实现的，而 Metal 增强组件是 Action Controller 实现的。

Middleware 是在请求进入 Controller#action 之前，而 Metal 增强组件是在请求进入 Controller#action 之后。

Middleware 需要的环境是 `@env`，作用的是 `app`；而 Metal 增强组件需要的环境是 Controller 和 action，目的主要是对请求做处理，并响应。

其它

更多关于 Action Controller

Controller 里的 public方法(也就是action) 会自动对应 Route 里的路由规则。当请求到来时，action 接受请求并处理，最后渲染相应视图模板(Get-and-show)或重定向到另一 action(do-and-redirect).

默认，只有 ApplicationController 直接继承于 ActionController::Base，其它的控制器继承于 ApplicationController. 所以，如果你想在所有 controller 处理之前做一些什么，你可以把它们写在 ApplicationController 里。

ActionController include 了对 metal/ 目录下面的模块，而我们自定义的 Controller 又继承于 ActionController.

自然的，它们的 ClassMethods 就会变成我们自定义 Controller 的类方法，而其它方法则类似实例方法，可运用于 action.

运行 action 时的魔法

请求从 Action Dispatch 的 Routes 里转发过来，首先到达 Metal 的 self.action 方法，然后经过层层 middleware 处理。
再然后调用 Metal 的 dispatch 方法(这里会创建 Metal 的实例对象)，调用 AbstractController::Base 的 process --> process_action --> send_action & send 完成。

Form Builder

`default_form_builder`

可以更改表单构造器。（表单本身也是一个对象，通过 Form Builder 添加更多方法）

```
class AdminFormBuilder < ActionView::Helpers::FormBuilder
  def special_field(name)
    # ...
  end
end
```

```
class AdminController < ApplicationController
  default_form_builder AdminFormBuilder
end
```

调用：

```
<%= form_for(@instance) do |builder| %>
  <%= builder.special_field(:name) %>
<% end %>
```

Renderer

让渲染模板需要的环境变得更容易，不必强制依赖于具体的 Controller#action.

```
ApplicationController.renderer.render template: '...'
# 更简短的写法
ApplicationController.render template: '...'

ApplicationController.renderer.new(method: 'post', https: true)
```

原来的那一套渲染流程你是需要依赖于某个 Controller 这大环境的，但有的场景并不需要（比如使用 Action Cable 时）。

所以，现在拆分出来了。你可以不用依赖于 Controller，这样你能很方便的在 Job、Script 及 web sockets 里调用/渲染模板。

`renderer` 获取渲染器实例：

```
ApplicationController.renderer
```

它所依赖的环境 `@env` 像 Rack 一样，非常简单：

```
ApplicationController.renderer.defaults
```

~~Railties Helpers~~

根据配置及其它因素，决定是否给我们所定义的 Controller 加载所有可用的 helpers.

也就是：

```
klass.helper :all
```

> Note: 可通过 config.action_controller.include_all_helpers 进行配置只加载 application_helper 和当前 Controller 所对应的 x_helper.

Action Dispatch Routing 顶层

提供接口，让我们定义应用相关路由。

先说说 Rack endpoint

Rack endpoint 是什么？
需要明确 Rack 是一个协议，符合这个协议的程序统称为 Rack application. Rack application 根据表现形式、调用方式、作用等又引申出几个概念，其中就包括 Rack endpoint. 在这里不作讨论和区分，统一对待。也就是说：

Rack ~= Rack middleware ~= Rack endpoint ~= Rack application

再说 Routing

一切路由规则都可归结为: 映射路径到 Rack endpoint.

这里的 Rack endpoint 指的不仅仅是 Controller#action，其它形式的入口也可以，例如：Engine、Sinatra 应用。

Routing 主要包含两部分：

Mapper 这部分，也就是路由机制这部分，这是我们接触得最多的，它包括：Base、Concerns、HttpHelpers、Resources、Scoping.

除了 Mapper 外，用到的还有：Redirection、Polymorphic Routes、Url For.

Routing 所在文件、目录：除 RouteSet、Routes Proxy 和 Journey 外，routing 目录里的其它模块。

Mapper

TODO

Base

常用方法：

```
match

mount

root
```

match 方法

这里的 `match` 只是个同名方法，是个空壳子，具体实现要看 Resources 里的 `match` .

另外，mount 和 root 本质上，都是封装和扩展 `match` 方法。

```
mount 和 root
      |
      V
    match
```

mount 方法

挂载一个基于 Rack 的应用到我们的程序。

```
match '/movies/search', => "movies#search"
# 去掉语法糖，等价
match '/movies/search', => MoviesController.action(:search)
```

```
# 这是一个 Rack. 所以可以调用 call 方法，传递 env 对象。
MoviesController.action(:search).call(env)

# 这也是一个 Rack.
Proc.new { |env|
  [
    200,
    {"Content-Type" => 'text/plain'},
    ["Hello, world"]
  ]
}

# 这还是一个 Rack.
class ApiApp
  def call(env)
    [
      200,
      {"Content-Type" => 'text/plain'},
      ["Hello, world"]
    ]
  end
end
```

```
# 替换 Rack
match '/movies/search', => proc { |env|
                                  [
                                    200,
                                    {"Content-Type" => 'text/pla
in'},
                                    ["Hello, world"]
                                  ]
                                }
# 替换 Rack
match '/movies/search', => ApiApp
```

```
# 进价
scope '/api' do
  match '(*path)', => ApiApp
end

# 加上语法糖，等价
mount ApiApp, :at => '/api'
# 或
mount ApiApp => "api"
```

因为 `mount` 实现基于 `match` ，可以使用相同的可选参数。例如：

```
mount(SomeRackApp => "some_route", as: "exciting")
```

现在，你可以通过 `exciting_path` 或 `exciting_url` 访问到刚才挂载的应用。

使用 `mount` 代替 `match` 还有一个细节不同，在被挂载的 Rack endpoint 里，映射路由时我们不必加前缀。举例：

不是：

```
require 'sinatra'
class ApiApp < Sinatra::Base
  get '/api' do
  end

  get "/api/endpoint" do
  end

  post "/api/endpoint" do
  end
end
```

而是：

```
require 'sinatra'
class ApiApp < Sinatra::Base
  get '/' do
  end

  get "/endpoint" do
  end

  post "/endpoint" do
  end
end
```

root 方法

实现：

```
def root(options = {})
  match '/', { :as => :root, :via => :get }.merge!(options)
end
```

因为 `root` 实现基于 `match`，可以使用相同的可选参数。

> 建议你把 `root` 放在 `config/routes.rb` 的开头部分，因为 Rails 的匹配规则是从上至下生成的，会优先匹配。

Http Helpers

match 方法的语法糖。

对应 HTTP 请求里的：

```
delete
get
patch
post
put
```

并且：

```
delete + get + patch + post + put
             |
             V
           match
```

使用上：

```
# 注意 via 参数
match 'photos/:id' => 'photos#show', via: :get

# 对应着
get 'photos/:id' => 'photos#show'
```

Scoping

scope

当几个路由规则的可选参数(部分)都一样的时候，可以用 `scope` 把它们包住，避免重复。

`scope` 并不会添加路由，但它会设置 @scope 的值，从而影响如何生成路由。

```
def scope(*args)
  # ... ...

  @scope.options.each do |option|
    # ...
  end

  @scope = @scope.new scope
  yield
  self
ensure
  @scope = @scope.parent
end
```

另，`@scope` 是 Scope 的实例对象。

以下几个方法，直接封装于它：

方法	解释
namespace	基于 scope，只是设置了 :module => path 本质是设置了 scope 的 :path、:module 和 :as 参数
controller	基于 scope，只是设置了 :controller => controller 和普通写法差不多，只是把单行改为了多行的形式
constraints	基于 scope，只是设置了 :constraints 传递的是限制条件，符合条件的请求才会进入里面的路由，进行下一步操作。
defaults	基于 scope，只是设置了 :defaults 作用是设置默认值

`scope` 可选参数：

```
SCOPE_OPTIONS = [:path, :shallow_path, :as, :shallow_prefix, :mo
dule,
                 :controller, :action, :path_names, :constraints
,
                 :shallow, :blocks, :defaults, :options]
```

`controller` 和 `constraints` 仅做约束条件，不影响路由的'一个萝卜，一个坑'的特性。其它几个方法，看情况有时会影响。

使用举例：

```
scope module: "admin" do
  resources :posts
end

# 或

resources :posts, module: "admin"
```

可将 "/posts" 路由到 Admin::PostsController

```
scope "/admin" do
  resources :posts
end

# 或

resources :posts, path: "/admin/posts"
```

可将 "/admin/posts" 路由到 PostsController

scope

一、不使用 scope

```
resources :jjs
```

默认对应：

```
        jjs GET    /jjs(.:format)                     jjs#index
            POST   /jjs(.:format)                     jjs#creat
e
     new_jj GET    /jjs/new(.:format)                 jjs#new
    edit_jj GET    /jjs/:id/edit(.:format)            jjs#edit
         jj GET    /jjs/:id(.:format)                 jjs#show
            PATCH  /jjs/:id(.:format)                 jjs#updat
e
            PUT    /jjs/:id(.:format)                 jjs#updat
e
            DELETE /jjs/:id(.:format)                 jjs#destr
oy
```

二、使用 scope，但不传递参数

```
scope "/admin" do
  resources :iis
end
```

默认对应：

```
         iis GET    /admin/iis(.:format)              iis#index
             POST   /admin/iis(.:format)              iis#creat
e
      new_ii GET    /admin/iis/new(.:format)          iis#new
     edit_ii GET    /admin/iis/:id/edit(.:format)     iis#edit
          ii GET    /admin/iis/:id(.:format)          iis#show
             PATCH  /admin/iis/:id(.:format)          iis#updat
e
             PUT    /admin/iis/:id(.:format)          iis#updat
e
             DELETE /admin/iis/:id(.:format)          iis#destr
oy
```

可以看出，只影响最外面的网址。

三、使用 scope，并传递参数

接受参数：path、constraints、shallow_path、shallow_prefix、defaults、as、module、controller 等。

1) **:as** 参数

3 个很重要的参数之一

```
# 只影响中间 helper 方法
scope as: "admin" do
  resources :ffs
end
```

影响中间层

```
    admin_ffs GET    /ffs(.:format)                          ffs#ind
ex
              POST   /ffs(.:format)                          ffs#cre
ate
 new_admin_ff GET    /ffs/new(.:format)                      ffs#new
edit_admin_ff GET    /ffs/:id/edit(.:format)                 ffs#edi
t
     admin_ff GET    /ffs/:id(.:format)                      ffs#sho
w
              PATCH  /ffs/:id(.:format)                      ffs#upd
ate
              PUT    /ffs/:id(.:format)                      ffs#upd
ate
              DELETE /ffs/:id(.:format)                      ffs#des
troy
```

2) **:path** 参数

3 个很重要的参数之一

```
# 只影响最外面的网址
scope path: "/admin" do
  resources :ggs
end
```

影响最外层

```
         ggs GET    /admin/ggs(.:format)                 ggs#index
             POST   /admin/ggs(.:format)                 ggs#creat
e
      new_gg GET    /admin/ggs/new(.:format)             ggs#new
     edit_gg GET    /admin/ggs/:id/edit(.:format)        ggs#edit
          gg GET    /admin/ggs/:id(.:format)             ggs#show
             PATCH  /admin/ggs/:id(.:format)             ggs#updat
e
             PUT    /admin/ggs/:id(.:format)             ggs#updat
e
             DELETE /admin/ggs/:id(.:format)             ggs#destr
oy
```

3) **:module** 参数

3 个很重要的参数之一

```
# 只影响最里面的 Controller#action
scope module: "admin" do
  resources :hhs
end
```

影响最里层

```
        hhs GET    /hhs(.:format)                      admin/hhs
#index
            POST   /hhs(.:format)                      admin/hhs
#create
     new_hh GET    /hhs/new(.:format)                  admin/hhs
#new
    edit_hh GET    /hhs/:id/edit(.:format)             admin/hhs
#edit
         hh GET    /hhs/:id(.:format)                  admin/hhs
#show
            PATCH  /hhs/:id(.:format)                  admin/hhs
#update
            PUT    /hhs/:id(.:format)                  admin/hhs
#update
            DELETE /hhs/:id(.:format)                  admin/hhs
#destroy
```

4) **:path、:as 和 :module 参数**

```
scope path: ":admin", as: "admin", module: 'admin' do
  resources :dds
end
```

等价于

namespace :admin

```
    admin_dds GET    /admin/dds(.:format)                 admin/d
ds#index
              POST   /admin/dds(.:format)                 admin/d
ds#create
 new_admin_dd GET    /admin/dds/new(.:format)             admin/d
ds#new
edit_admin_dd GET    /admin/dds/:id/edit(.:format)        admin/d
ds#edit
     admin_dd GET    /admin/dds/:id(.:format)             admin/d
ds#show
              PATCH  /admin/dds/:id(.:format)             admin/d
ds#update
              PUT    /admin/dds/:id(.:format)             admin/d
ds#update
              DELETE /admin/dds/:id(.:format)             admin/d
ds#destroy
```

5) **:defaults** 参数

对应着 defaults 方法。(它是通用的，类型为 Hash.)

6) **:constraints** 参数

对应着 constraints 方法。

7) **:controller** 参数

对应着 controller 方法。

namespace

```
namespace :admin do
  resources :posts
end
```

默认对应：

```
    # 中间层 helper 方法  # 最外层网址                    # 最里层 C
ontroller#action
    admin_posts GET       /admin/posts(.:format)          admin/
posts#index
    admin_posts POST      /admin/posts(.:format)          admin/
posts#create
 new_admin_post GET       /admin/posts/new(.:format)      admin/
posts#new
edit_admin_post GET       /admin/posts/:id/edit(.:format) admin/
posts#edit
     admin_post GET       /admin/posts/:id(.:format)      admin/
posts#show
     admin_post PATCH/PUT /admin/posts/:id(.:format)      admin/
posts#update
     admin_post DELETE    /admin/posts/:id(.:format)      admin/
posts#destroy
```

1) 使用 `:path` 更改最外层网址：

```
# 使用 /sekret/posts 而不是 /admin/posts
namespace :admin, path: "sekret" do
  resources :posts
end
```

2) 使用 `:module` 更改最里层 Controller#action：

```
# 使用 Sekret::PostsController 而不是 Admin::PostsController
namespace :admin, module: "sekret" do
  resources :posts
end
```

3) 使用 `:as` 更改中间层 helper 方法：

```
# 使用 sekret_posts_path 而不是 admin_posts_path
namespace :admin, as: "sekret" do
  resources :posts
end
```

4) 此外：

```
namespace "admin" do
  resources :kks
end

# 等价于

scope module: "admin", path: "/admin", as: "admin" do
  resources :kks
end
```

Concerns

重复使用已经定义的路由，避免 `config/routes.rb` 里出现重复代码，和生成路由规则没有直接联系。

方法	解释
concern	定义(一次只能定义一个)
concerns	调用(一次可调用一个或多个)

举例(使用 concern & concerns 之前)：

```
AppName::Application.routes.draw do
  resources :messages do
    resources :comments
    resources :categories
    resources :tags
  end

  resources :posts do
    resources :comments
    resources :categories
    resources :tags
  end

  # ...
end
```

举例(使用 concern & concerns 之后)：

```
AppName::Application.routes.draw do
  concern :sociable do
    resources :comments
    resources :categories
    resources :tags
  end

  resources :messages do
    concerns :sociable
  end

  resources :posts do
    concerns :sociable
  end

  # ...
end
```

再次举例：

```
concern :commentable do
  resources :comments
end

concern :image_attachable do
  resources :images, only: :index
end

# concerns 可以跟多个 concern
resources :messages, concerns: [:commentable, :image_attachable]

namespace :posts do
  concerns :commentable
end
```

一些疑问？

即使是去除重复代码，也还有其它方法实现。如：

```
AppName::Application.routes.draw do
  def add_posts
    resources :posts, :only => [:create, :destroy]
  end

  resources :events do
    add_posts
  end
end
```

有必要使用 DSL 吗？

在 讨论 里 dhh 回答了此问题。

Resources

常用方法：

方法	解释
resource	我们的资源不需要集合操作的时候可以使用
resources	我们的资源需要集合操作的时候可以使用，本方法最常用
match	生成路由规则，并添加到 @set 里

resource

单个资源。

调用了 collection 和 new，以及 set_member_mappings_for_resource 完成所有。

```
      resource
         |
         V
collection 和 new 和 member
         |
         V
       scope
```

在这里：`colleciton` 完成 create；`new` 完成 new；`member` 完成 edit、show、update 和 destroy.

resources

多个资源。

调用了 collection 和 new，以及 set_member_mappings_for_resource 完成所有。

```
      resources
          |
          v
collection 和 new 和 member
          |
          v
        scope
```

在这里：`colleciton` 完成 index 和 create；`new` 完成 new；`member` 完成 edit、show、update 和 destroy.

match

匹配 url 到一个或多个路由。所有符号，都会对应着 url 里的参数，可用 `params` 获取：

```
# 对应着 params 里的 :controller, :action 和 :id
match ':controller/:action/:id'
```

预留了两个符号，`:controller` 对应着 Controller，`:action` 对应着 action. 也可以接受模式匹配做为参数：

```
# 路由
match 'songs/*category/:title', to: 'songs#show'

# URL
'songs/rock/classic/stairway-to-heaven'

# 对应：
params[:category] = 'rock/classic'
params[:title] = 'stairway-to-heaven'
```

为了能够模式匹配，你需要分配一个名字给它们，如果没有分配，路由是不会自动解析的。

使用模式匹配时，路由里的 :action 和 :controller 应该以 Hash 的形式传递过来比较好。例如：

```
match 'photos/:id' => 'photos#show'
match 'photos/:id', to: 'photos#show'
match 'photos/:id', controller: 'photos', action: 'show'
```

模式匹配，也可以直接指向 Rack application. 因为它实现了 `call` 方法：

```
match 'photos/:id', to: lambda {|hash| [200, {}, ["Coming soon"]
] }
match 'photos/:id', to: PhotoRackApp

# YourController.action(:your_action) 也是 rack endpoint
match 'photos/:id', to: PhotosController.action(:show)
```

通过 HTTP 请求，容易带来安全隐患，所以你可以使用 HtttpHelpers[rdoc-ref:HttpHelpers]，而不是 `match`

另：

```
   match
     |
     V
decomposed_match (还分几种情况)
     |
     V
  add_route
     |
     V
@set.add_route
```

其它：

其它方法：

```
nested 和 root
       |
       V
     scope
```

```
namespace
   |
   V
 super (即 Scoping 里的 namespace)
   |
   V
 scope
```

```
resources_path_names

shallow
shallow?

using_match_shorthand?
```

protected 方法：

```
set_member_mappings_for_resource

with_exclusive_scope

with_scope_level
```

`set_member_mappings_for_resource` 我们路由里的 edit、show、update 和 destroy 由它完成。

还有：

```
VALID_ON_OPTIONS = [:new, :collection, :member]
RESOURCE_OPTIONS = [:as, :controller, :path, :only, :except, :pa
ram, :concerns]
```

包含：~~Scope、Mapping、Constraints~~

~~Scope~~

`@scope` 是其实例对象。

```
module ActionDispatch
  module Routing
    class Mapper
      def initialize(set)
        @set = set
        @scope = Scope.new({ :path_names => @set.resources_path_
names })
        @concerns = {}
        @nesting = []
      end
    end
  end
end
```

~~Mapping~~

标准化路由规则。

match 会调用 add_route，进而 @set.add_route 完成添加路由规则。但在 @set.add_route 之前，要先把路由规则标准化。

```ruby
module ActionDispatch
  module Routing
    class Mapper
      module Resources
        # ...

        def match
          # ...
        end

        # ... ...

        def add_route(action, options)
          path = path_for_action(action, options.delete(:path))

          # ... ...

          as = if !options.fetch(:as, true)
                 options.delete(:as)
               else
                 name_for_action(options.delete(:as), action)
               end

          mapping = Mapping.build(@scope, @set, URI.parser.escape(path), as, options)
          app, conditions, requirements, defaults, as, anchor = mapping.to_route

          @set.add_route(app, conditions, requirements, defaults, as, anchor)
        end
      end
    end
  end
end
```

~~Constraints~~

标准化路由规则这个过程中，涉及到的一个对象。

```
module ActionDispatch
  module Routing
    class Mapper
      class Mapping
        def to_route
          [ app(@blocks), conditions, requirements, defaults, as
, anchor ]
        end

        private
        def app(blocks)
          if to.respond_to?(:call)
            Constraints.new(to, blocks, false)
          elsif blocks.any?
            Constraints.new(dispatcher(defaults), blocks, true)
          else
            dispatcher(defaults)
          end
        end
      end
    end
  end
end
```

Endpoint 的子类之一，它是 endpoint.

Redirection

路由里的 **redirect** 方法。

`redirect(*args, &block)`

在 routes.rb 里配置重定向，将发向某路径的请求，重定向到另一路径：

```
get "/stories" => redirect("/posts")
```

你也可以动态的重定向到新的路径：

```
get 'docs/:article', to: redirect('/wiki/%{article}')
```

这里的重定向，在路由里完成，不经过 Action Controller 等处理。
并且，在浏览器里会看到 url 的改变。

相关类有：OptionRedirect、PathRedirect 及 Redirect. 根据不同的参数情况，会对应的使用它们做处理。在这里我们只关心结果，不必太在意细节。

match 和 scope 方法 - 重中之重

从前面各章节，可归纳出路由规则里， match 和 scope 两方法是重中之重。
其它方法，都在在这两方法的基础上，做封装、语法糖、去重复代码等。

`match` 是生成路由规则， `scope` 是影响如何生成路由规则。

和 `match` 有关的有：get, post, delete, put 等, 只有通过它们才能添加路由规则；
和 `scope` 有关的有：namescope, collection, member, resource, resouces 及嵌套路由时自动完成的 nested 等。它们主要工作是调用上面的 match 相关方法，并影响生成路由规则。

完成路由规则整个过程中，match 和 scope 缺一不可。

路由常用方法汇总

下面方法都可以看做是 `Mapper` 的实例方法，如果我们想要定制自己的方法，参考它们即可。

基本：

```
root

mount

match
```

Http 方法：

```
get
post
patch
put
delete
```

重定向：

```
redirect
```

作用域：

```
scope
controller
namespace
constraints
defaults
```

关联：

```
concern
concerns
```

资源：

```
resource
resources
collection
member
```

常用 gem 'devise' 就是通过给 `Mapper` 增加实例方法的方式，在路由里为我们提供了 `devise_for` 和 `devise_scope` 方法。

其它

TODO

Polymorphic Routes

多态路由辅助方法。

实例方法

```
polymorphic_url
polymorphic_path
```

1. 不仅仅是多态关联里的'多态'
2. 可根据参数(record 对象)，自动计算生成 url
3. 有几个常用方法是基于它实现的
4. 不用指定具体的路由 helper 方法
5. 对象类型不确定或嵌套对象
6. 同样依赖于路由系统
7. 使用它会使得复杂度提高，难以理解，所以不要滥用

使用之前的做法：

```
# 在这里 parent 可以是 post 或 news
if Post === parent
  post_comments_path(parent)
elsif News === parent
  news_comments_path(parent)
end
```

使用之后：

```
# 使用 post_url(post)
polymorphic_url(post)                  # => "http://example.com/posts/1"
polymorphic_url([blog, post])          # => "http://example.com/blogs/1/posts/1"
polymorphic_url([:admin, blog, post]) # => "http://example.com/admin/blogs/1/posts/1"
polymorphic_url([user, :blog, post])  # => "http://example.com/users/1/blog/posts/1"
polymorphic_url(Comment)               # => "http://example.com/comments"
```

> Note：比较常用的方式是以数组的形式传递参数。

当然，多态关联也可用：

```
class Post < ActiveRecord::Base
  has_many :comments
end
class News < ActiveRecord::Base
  has_many :comments
end
class Comment < ActiveRecord::Base
  belongs_to :commentable, :polymorphic => true
end
```

```
polymorphic_path([parent, Comment])
# "/posts/1/comments"
# 或
# "news/1/comments"

polymorphic_url(parent)
# "http://example.com/posts/1/comments"
# 或
# "http://example.com/news/1/comments"

其它
new_polymorphic_path(Post)  # "/posts/new"
new_polymorphic_url(Post)   # "http://example.com/posts/new"
edit_polymorphic_path(post) # "/posts/1/edit"
edit_polymorphic_url(post)  # "http://example.com/posts/1/edit"
```

:action 以及其它参数

举例：

```
# 使用 :aciton 可选参数
polymorphic_path([@user, Document], :action => 'filter')
# => "/users/:user_id/documents/filter"

# 使用 :aciton 可选参数和其它参数
polymorphic_path([@user, Document], :action => 'filter', :sort_o
rder => 'this-order')
# => "/users/:user_id/documents/filter?sort_order=this-order"
```

与 url_for 的区别

url_for 比较死板，从它接受的参数就知道了。它不能接受 record 对象 + 可选参数的形式。

url_for 不能直接指定 host, 需要在另一个地方指定，它只有调用的份。如果你为了一个 url_for 而更改这个 host 其它方法或其它 url_for 会不会受影响？

其它方法

除上述外，还有方法(元编程生成，API 里查看不到)：

```
# 封装 polymorphic_url 而来
new_polymorphic_url
edit_polymorphic_url

# 封装 polymorphic_path 而来
new_polymorphic_path
edit_polymorphic_path
```

它们封装 polymorphic_url 或 polymorphic_path 而来，所以特点和使用类似。 它们是元编程定义的，所以 API 里看不到。

~~Helper Method Builder~~

以字符串拼接为手段，得到具体的路由 helper 方法(不能直接得到网址！)。后续，可以再通过路由 helper 方法，得到网址。

:action - 其实是生成的方法所带的前缀，默认是没有的。Rails 使用了前缀 :new 和 :edit

:routing_type - 生成 :path 还是 :url, 默认是 :url.

```
ActionDispatch::Routing::PolymorphicRoutes::HelperMethodBuilder.
plural 'edit', 'url'

when Array
builder.handle_list record   # 拆分处理，方式雷同
  when String, Symbol
builder.handle_string record # 直接使用此字符串
  when Class
builder.handle_class record  # 小写，复数形式
  else
builder.handle_model record  # 类型，小写，单数形式
```

现在可以看到 Action View 和 Action Dispatch 里的 url_for 方法, 以及 Action Dispatch 里的 Polymorphic Routes 相关方法都封装了它，在某些参数情况下会调用到它。其它地方，没有使用。

其它

原来这个模块是在 Action Controller 下面的，后面才移到 ActionDispatch::Routing.

我们是可以直接使用这几个方法的。

Url For

路由里的 **url_for** 方法。

和 ActionView::RoutingUrlFor 原理一样，封装了 Helper Method Builder. (极端情况下才调用到)

使用举例：

```
url_for controller: 'tasks', action: 'testing', host: 'example.o
rg', port: '8080'
# => 'http://example.org:8080/tasks/testing'

url_for controller: 'tasks', action: 'testing', host: 'example.o
rg',
        anchor: 'ok', only_path: true
# => '/tasks/testing#ok'

url_for controller: 'tasks', action: 'testing', trailing_slash:
true
# => 'http://example.org/tasks/testing/'

url_for controller: 'tasks', action: 'testing', host: 'example.o
rg', number: '33'
# => 'http://example.org/tasks/testing?number=33'

url_for controller: 'tasks', action: 'testing', host: 'example.o
rg',
        script_name: "/myapp"
# => 'http://example.org/myapp/tasks/testing'

url_for controller: 'tasks', action: 'testing', host: 'example.o
rg',
        script_name: "/myapp", only_path: true
# => '/myapp/tasks/testing'
```

可选参数的类型可以是：nil、Hash、String、Symbol、Array、Class 等，根据不同参数，可能会用到【Polymorphic Routes】Helper Method Builder 里的东西。

Routing 概述：生成、存储、识别

描述

1. DSL --> Mapper(Base, Concerns, HttpHelpers, Resources, Scoping)
2. 既然维护着一张路由表，如何向表里添加规则？
3. 外部的 url 是如何识别并处理的？先预处理，然后对照路由表，然后转发给 Controller#action，再接下来就是我们熟悉的东西了。
4. 路由分类具名路由 和 匿名路由，内部可以调用具名路由(不调用，它就没意义了)。那么这些方法在哪定义的？
5. Rails 引入了 engine 的概念，涉及到 engine 的路由表是如何工作的？ 注意上面描述里的动词

1) 里对应着 mapper.rb 文件，及里面的各个子模块。

2) 由 add_route 完成，涉及一大堆的东西。

3) 里的转发给 Controller#action 这部分，由 Dispatcher 完成。通过 controller.action(action).call(env)，到具体的 Controller & action -- 常见的 rack 用法。

4) 里的定义方法，由 Named Route Collection 完成。它还代表了具名路由。

```
def define_url_helper(route, name, options)
  helper = UrlHelper.create(route, options.dup)

  @module.remove_possible_method name
  @module.module_eval do
    define_method(name) do |*args|
      helper.call self, args
    end
  end

  helpers << name
end

def define_named_route_methods(name, route)
  define_url_helper route, :"#{name}_path",
    route.defaults.merge(:use_route => name, :only_path => true)
  define_url_helper route, :"#{name}_url",
    route.defaults.merge(:use_route => name, :only_path => false
)
end
```

4) 内部调用并不一定总是通过 具体路由进行调用，例如：有时候我们会使用 link_to @record ... 这种情况，极端情况下会由路由这边生成 url，由 Generator 完成。

5) 涉及 engine 的路由部分，由 Mounted Helpers 完成。

Journey 就是个打杂的，其它看得见和看不见的功能由它负责。

路由的生成、存储、识别

1) 路由对象的定义和调用方式：

```
# 定义
module Rails
  class Engine < Railtie
    def routes
      @routes ||= ActionDispatch::Routing::RouteSet.new
      @routes.append(&Proc.new) if block_given? # 调用时，也可以追加路由规则
      @routes
    end
  end
end

# 调用方式
Rails.application.routes
Rails.application.routes { ... }
```

2) 调用路由对象，生成路由规则：

```
Rails.application.routes.draw do
  # ... block 内容
end
```

3) 生成路由规则，本质是对 block 内容的求值：

```
def draw(&block)
  # ...
  eval_block(block)
  # ...
  nil
end
```

4) 怎么求值呢？借助了 Mapper 的实例对象：

```
def eval_block(block)
  # ...
  mapper = Mapper.new(self)
  # ...
  mapper.instance_exec(&block)
end
```

5) Mapper 的实例对象有什么内容？

```
module ActionDispatch
  module Routing
    class Mapper
      # 这里 set = Rails.application.routes
      def initialize(set)
        @set = set
        @scope = Scope.new({ :path_names => @set.resources_path_
names })
        @concerns = {}
        @nesting = []
      end
    end
  end
end
```

6) Mapper 实例对象有了，下一步就是执行 instance_exec. 也就是运行 block 里的各个方法。

7) block 里的各个方法，是在 Mapper 下面的各个模块里定义的：

```
module ActionDispatch
  module Routing
    class Mapper
      # ...

      class Constraints < Endpoint
        # ...
      end
```

```
    class Mapping
      # ...
    end

    class Scope
      # ...
    end

    # ... 下面的各个模块

    module Base
      # ...
    end

    module HttpHelpers
      # ...
    end

    module Redirection
      # ...
    end

    module Scoping
      # ...
    end

    module Concerns
      # ...
    end

    module Resources
      # ...
    end

    include Base
    include HttpHelpers
    include Redirection
    include Scoping
    include Concerns
    include Resources
```

```
    end
  end
end
```

在这里，不同的路由规则会有对应的模块进行处理，具体可以在对应的各个章节查看。

8) 附：Mapper 的 ancestors

```
ActionDispatch::Routing::Mapper.ancestors
=> [ActionDispatch::Routing::Mapper,

 ActionDispatch::Routing::Mapper::Resources,
 ActionDispatch::Routing::Mapper::Concerns,
 ActionDispatch::Routing::Mapper::Scoping,
 ActionDispatch::Routing::Redirection,
 ActionDispatch::Routing::Mapper::HttpHelpers,
 ActionDispatch::Routing::Mapper::Base,

 # ...,
 ...]
```

一步步分析从请求到响应涉及到 Rails 的哪些模块

通过本章节，你可以比较独立的使用 Rails 的各个模块，有利于理解 Rails 源码各个模块的主要工作。

纯 Rack 实现

```
# config.ru
require 'bundler/setup'

run Proc.new {|env|
 if env["PATH_INFO"] == "/"
   [200, {"Content-Type" => "text/html"}, ["<h1>Hello World</h1>"
]]
 else
   [404, {"Content-Type" => "text/html"}, ["<h1>Not Found</h1>"]
]
 end
}
```

可通过 `rackup config.ru` 运行以上代码，默认在 http://localhost:9292/ 可以查看运行结果。

引入 Action Dispatch & 纯手动实现 Controller#actions

```
# config.ru
require 'bundler/setup'
require 'action_dispatch'

routes = ActionDispatch::Routing::RouteSet.new
routes.draw do
  get '/' => 'mainpage#index'
  # 和以下写法效果一样，但'这里'要把它定义在 MainpageController 后面
  # get '/' => MainpageController.action("index")

  get '/page/:id' => 'mainpage#show'
end

class MainpageController
  def self.action(method)
    controller = self.new
    controller.method(method.to_sym)
  end

  def index(env)
    [200, {"Content-Type" => "text/html"}, ["<h1>Front Page</h1>"
]]
  end

  def show(env)
    [200, {"Content-Type" => "text/html"},
     ["<pre> #{env['action_dispatch.request.path_parameters'][:id
]} #</pre>"]]
  end
end

run routes
```

引入 Action Controller，使用 Metal

```
# config.ru
require 'bundler/setup'
require 'action_dispatch'
require 'action_controller'

routes = ActionDispatch::Routing::RouteSet.new
routes.draw do
  get '/' => 'mainpage#index'
  get '/page/:id' => 'mainpage#show'
end

class MainpageController < ActionController::Metal
  def index
    self.response_body = "<h1>Front Page</h1>"
  end

  def show
    self.status = 404
    self.response_body =
      "<pre>#{env['action_dispatch.request.path_parameters'] [:id
]}</pre>"
  end
end

run routes
```

注意：上面已经使用到了 Action Dispatch 里的"各个 Middleware 组件"，但并没有使用到 Action Controller 里的"各个 Metal 增强组件"。

引用 Metal 增强模块 & Controller 里纯手工打造 View 渲染相关代码

```
# config.ru
require 'bundler/setup'
require 'action_dispatch'
require 'action_controller'

routes = ActionDispatch::Routing::RouteSet.new
routes.draw do
  get '/' => 'mainpage#index'
  get '/page/:id' => 'mainpage#show'
end

class MainpageController < ActionController::Metal
  include AbstractController::Rendering
  include ActionController::Rendering
  include ActionController::ImplicitRender

  def index
    # self.response_body = "<h1>Front Page</h1>"
    @local_var = 12345
  end

  def show
    self.status = 404
    self.response_body =
      "<pre>#{env['action_dispatch.request.path_parameters'] [:id
]}</pre>"
  end

  def render_to_body(*args)
    template = ERB.new File.read("#{params[:action]}.html.erb")
    template.result(binding)
  end
end

run routes
```

```
# index.html.erb

Number is: <%= @local_var %>
```

引进 Action View

```
# config.ru
require 'bundler/setup'
require 'action_dispatch'
require 'action_controller'
require 'action_view'

routes = ActionDispatch::Routing::RouteSet.new
routes.draw do
  get '/' => 'mainpage#index'
  get '/page/:id' => 'mainpage#show'
end

class MainpageController < ActionController::Metal
  include AbstractController::Rendering
  include ActionView::Rendering
  include ActionController::Rendering
  include ActionController::ImplicitRender

  prepend_view_path('app/views')

  def index
    @local_var = 12345
  end

  def show
  end
end

use ActionDispatch::DebugExceptions
run routes
```

```
# app/views/mainpage/index.html.erb

Number is: <%= @local_var %>
```

```
# app/views/mainpage/show.html.erb

Content is: <pre><%= env['action_dispatch.request.path_parameters'][:id] %></pre>
```

到此，Routing、Controller、View 3者已经到位。Model 我们可以直接调用，不影响这里的整个请求、响应处理过程。

直接使用 ActionController::Base

前面提到过 ActionController::Metal 相当于一个加强版本的 Rack，功能非常有限，实际开发中建议使用继承于它的 ActionController::Base.

```
# config.ru
# require 'bundler/setup'
# require 'action_dispatch'
# require 'action_view'
require 'action_controller'

routes = ActionDispatch::Routing::RouteSet.new
routes.draw do
  get '/' => 'mainpage#index'
  get '/page/:id' => 'mainpage#show'
end

class MainpageController < ActionController::Base
  prepend_view_path('app/views/')

  def index
    @local_var = 12345
  end

  def show
  end
end

use ActionDispatch::DebugExceptions
run routes
```

对应以上代码，我们需要创建视图文件。

```
# app/views/mainpage/index.html.erb
```

和

```
# app/views/mainpage/show.html.erb
```

其它

上述代码：
没有使用 Action Mailer，Active Job, Active Model, Active Record 等；
使用但没感受到 Abstract Controller，Active Support, Railties 等；
没有使用 Engine 等。

上面代码里的 `run` 方法由应用服务器提供，用于运行一个 Rack application.

参考

A Rails App in a Single File

~~endpoint 和 inspector 文件~~

endpoint 文件下的内容：

类	解释
Endpoint	Mapper::Constraints、 Redirect 和 RouteSet::Dispatcher 的抽象类。 定义了 dispatcher?、redirect?、matches? 和 app 方法。

inspector 文件下的内容：

类	解释
Route Wrapper	被 Routes Inspector 调用。
Routes Inspector	被 middleware Debug Exceptions 和 rake routes 调用。
Html Table Formatter	格式化 HTML 里的路由信息，给人阅读的。
Console Formatter	格式化控制台里的路由信息，给人阅读的。

Action Dispatch Middleware

Middleware - 在路由转发之后，**Controller#action** 接收之前，对环境和应用进行处理。

路由转发(Dispatcher#dispatch) --> 中间地带(middleware) --> 控制器(Controller#action)。

已经进入 Rails, 但还没正式进入我们应用，这时候需要做一些事。

Web 服务器已经处理过，转发过来了。做一些什么事呢？看一看下面的 middleware 都提供了什么功能就知道了。

下面只是做简介，详情可以查看 API 说明，或阅读源代码。

Middleware

TODO

Rack - Ruby Web server 接口

用 Ruby 或"Ruby 实现的 Web 框架"创建的应用，想要提供 Web 服务，它们都要和 Web 服务器打交道。而这个过程是非常复杂的，为了简化这个过程，我们可以使用 Rack.

> app应用 --> (Rack) --> 应用服务器 --> Web服务器 --> 外部世界

Rack 提供了一个"与 Web服务器打交道"最精简的接口，通过这个接口，我们的应用很轻松的就能提供 Web 服务(接收 Web 请求，响应处理结果)。上面的"应用服务器"是对 Rack 的进一步封装。

使用这个接口的条件是：传递一个"程序"(你没看错，就是把一个程序当做参数，下文以 app 代替)。并且这个 app 需要同时满足以下条件：

- `app.respond_to? :call` # => true
- app 创建过程需要以运行环境做为参数(类型为 Hash，下文以 `env` 代替运行环境)
- app 需要返回一个数组，数组包含 3 个元素，依次是：
    - HTTP status code
    - HTTP headers (类型为 Hash)
    - HTTP body data (类型能接受 `each` 方法即可)

最小的 Rack

```
Proc.new { |env|
  [
    200,
    {"Content-Type" => 'text/plain'},
    ["Hello, world"]
  ]
}
```

使用举例

```
# my_rack_app.rb

require 'rack'

app = Proc.new do |env|
  ['200', {'Content-Type' => 'text/html'}, ['A example rack app.'
]]
end

Rack::Handler::WEBrick.run app
```

你也可以使用 `rackup` 命令，节省点时间和力气：

```
# config.ru

run Proc.new { |env| ['200', {'Content-Type' => 'text/html'}, ['
get rack\'d']] }
```

运行:

`rackup config.ru`

... 完。

再举个例子：

可以封装 Rack，得到我们自己的 YourRack，或：

```
class YourRack
  def initialize(app)
    @app = app
  end

  def initialize(app)
    @app = app
  end

  def call(env)
    @app.call(env)
  end
end
```

再使用举例

'rack/contrib' 可以自动加载 Rack 包含的所有组件，以下例子可用 `rackup config.ru` 运行：

```
require 'rack'
require 'rack/contrib'

use Rack::Profiler if ENV['RACK_ENV'] == 'development'

use Rack::ETag
use Rack::MailExceptions

run theapp
```

链接

Rack: a Ruby Webserver Interface
a modular Ruby webserver interface
Contributed Rack Middleware and Utilities

查看项目用了哪些 Middleware

命令行里查看，项目有哪些 middleware:

```
rails middleware
```

如下：

```
use Rack::Sendfile

use ActionDispatch::Static
use ActionDispatch::LoadInterlock

use ActiveSupport::Cache::Strategy::LocalCache::Middleware

use Rack::Runtime
use Rack::MethodOverride

use ActionDispatch::RequestId

use Rails::Rack::Logger

use ActionDispatch::ShowExceptions

use WebConsole::Middleware

use ActionDispatch::DebugExceptions
use ActionDispatch::RemoteIp
use ActionDispatch::Reloader
use ActionDispatch::Callbacks

use ActiveRecord::Migration::CheckPending
use ActiveRecord::ConnectionAdapters::ConnectionManagement
use ActiveRecord::QueryCache

use ActionDispatch::Cookies
use ActionDispatch::Session::CookieStore
use ActionDispatch::Flash

use Rack::Head
use Rack::ConditionalGet
use Rack::ETag

use ActionView::Digestor::PerRequestDigestCacheExpiry

run AppName::Application.routes
```

控制台里查看，项目有哪些 middleware:

```
Rails.application.send :default_middleware_stack
```

如下：

```
    Rack::Sendfile,

    ActionDispatch::Static,
    ActionDispatch::LoadInterlock,

    Rack::Runtime,
    Rack::MethodOverride,

    ActionDispatch::RequestId,

    Rails::Rack::Logger,

    ActionDispatch::ShowExceptions,
    ActionDispatch::DebugExceptions,
    ActionDispatch::RemoteIp,
    ActionDispatch::Reloader,
    ActionDispatch::Callbacks,
    ActionDispatch::Cookies,
    ActionDispatch::Session::CookieStore,
    ActionDispatch::Flash,

    Rack::Head,
    Rack::ConditionalGet,
    Rack::ETag
```

```
Rails.application.send :middleware
```

如下：

```
Rack::Sendfile,

ActionDispatch::Static,
ActionDispatch::LoadInterlock,

ActiveSupport::Cache::Strategy::LocalCache::Middleware,

Rack::Runtime,
Rack::MethodOverride,

ActionDispatch::RequestId,

Rails::Rack::Logger,

ActionDispatch::ShowExceptions,

WebConsole::Middleware,

ActionDispatch::DebugExceptions,
ActionDispatch::RemoteIp,
ActionDispatch::Reloader,
ActionDispatch::Callbacks,

ActiveRecord::Migration::CheckPending,
ActiveRecord::ConnectionAdapters::ConnectionManagement,
ActiveRecord::QueryCache,

ActionDispatch::Cookies,
ActionDispatch::Session::CookieStore,
ActionDispatch::Flash,

Rack::Head,
Rack::ConditionalGet,
Rack::ETag,

ActionView::Digestor::PerRequestDigestCacheExpiry
```

说明：默认添加 middleware 可以在 Rails::Application::DefaultMiddlewareStack#build_stack 里查看；而我们手动添加的 middleware 通常在 config/ 下添加。执行middleware 由 ActionController::Metal.action 完成。

在上面例子里，除默认 middleware 外，我们还额外使用了 4 个 middleware. 在实际项目中，你应该可以看到比这多得多的 middleware.

另外，从应用的日常报错上，也能看出应用所使用的 middleware 及其顺序，在此就不贴示例了。

定制 Middleware

middleware 本质就是 **Rack app**，只是简单封装了一下。

我们可以编写自己的 Middleware，用来处理 `@app` 和 `env` .

举例一

放在 app/middleware/ 目录下，按照 middleware 写。然后在 config 里 use 就行了，不用做其它的配置。

```
class Scrubber
  def initialize(app, options)
    @app = app
    @routes = options[:routes]
  end

  def call(env)
    scrub(env)
    @app.call(env)
  end

  private
    def scrub(env)
      return unless @routes.include?(env["PATH_INFO"])
      rack_input = env["rack.input"].read
      params = Rack::Utils.parse_query(rack_input, "&")
      params["xml"] = Rack::Utils.unescape(params["xml"])
      env["rack.input"] = StringIO.new(Rack::Utils.build_query(p
arams))
    rescue
    ensure
      env["rack.input"].rewind
    end
end
```

然后：

```
# 注意，这里直接引用相关类
config.middleware.insert_before ActionController::ParamsParser,
                                "Scrubber",
                                :routes => [ "/examples/scrubbed" ]
```

举例二

放在 lib/ 目录下，按照 middleware 写。然后在 config 里 use 就行了，不用做其它的配置。

```
class ResponseTimer
  def initialize(app, message = "Response Time")
    @app = app
    @message = message
  end

  def call(env)
    dup._call(env)
  end

  def _call(env)
    @start = Time.now
    @status, @headers, @response = @app.call(env)
    @stop = Time.now
    [@status, @headers, self]
  end

  def each(&block)
    if @headers["Content-Type"].include? "text/html"
      block.call("<!-- #{@message}: #{@stop - @start} -->\n")
    end

    @response.each(&block)
  end
end
```

然后：

```
# 注意，这里以字符串的方式引用
config.middleware.use "ResponseTimer", "Load Time"
```

参考

Sanitizing POST params with custom Rack middleware

~~Middleware Stack~~

用实例变量 `@middlewares` 存储我们项目要使用的 middleware.

提供方法：

```
[]

build

delete

each

initialize_copy

insert & insert_before
insert_after

last

size

swap

unshift

use
```

> Note: 注意和【Configuration Middleware Stack Proxy】章节的联系与区别。

各个 Middleware 类

TODO

Static

从硬盘直接返回静态文件的内容。

默认这些静态文件存放在 public/ 目录下，如果找不到文件，或者移除此 middleware，则处理方式和普通请求一样，由路由决定转发到哪。

SSL

如果网站采用了 HSTS(HTTP Strict Transport Security) 协议，该 middleware 保证通过浏览器访问时，始终连接到该网站的 HTTPS 加密版本(也就是，访问被强制 https 了)，不需要用户手动在 URL 地址栏中输入加密地址。

实际项目中，如果用不到 HSTS(HTTP 严格传输安全协议)，可以考虑将此 middleware 移除。

Show Exceptions

程序抛异常时，用什么程序来处理。

它只是接口，它关心的是用什么，而不是如何处理。(因为不是它处理的！) 对内调用 Exception Wrapper 对外调用 Public Exceptions.

```
# 对内
wrapper = ExceptionWrapper.new(env, exception)
```

```
# 对外
config.exceptions_app || ActionDispatch::PublicExceptions.new(Rails.public_path)
response = @exceptions_app.call(env)
```

Exception Wrapper

更友好的异常消息，包括可读性，回朔等。

被 Show Exceptions 调用。(另，它其实不是 middleware)

```
ActionDispatch::ExceptionWrapper.rescue_responses

=> {"ActionController::RoutingError"=>:not_found,
  "AbstractController::ActionNotFound"=>:not_found,
  "ActionController::MethodNotAllowed"=>:method_not_allowed,
  "ActionController::NotImplemented"=>:not_implemented,
  "ActionController::InvalidAuthenticityToken"=>:unprocessable_e
ntity,
  "ActiveRecord::RecordNotFound"=>:not_found,
  "ActiveRecord::StaleObjectError"=>:conflict,
  "ActiveRecord::RecordInvalid"=>:unprocessable_entity,
  "ActiveRecord::RecordNotSaved"=>:unprocessable_entity}
```

```
ActionDispatch::ExceptionWrapper.rescue_templates

=> {"ActionView::MissingTemplate"=>"missing_template",
  "ActionController::RoutingError"=>"routing_error",
  "AbstractController::ActionNotFound"=>"unknown_action",
  "ActionView::Template::Error"=>"template_error"}
```

Public Exceptions

负责'渲染'错误页面这个工作。默认从 public/ 目录中寻找对应的异常文件，如 500 则渲染 /public/500.html

它只处理渲染相关逻辑，只是"定义"，至于要不要使用它做为默认异常处理，由 Show Exceptions 和 Exception Wrapper 等共同决定。

定制 Public Exceptions

默认的 Public Exceptions，显示 public/ 目录下的错误页面。

`config.exceptions_app` 可以配置 Show Excepting 抛异常时如何处理。如：

举例一：

```
# config/environments/production.rb
config.exceptions_app = ErrorController.action(:handle)
```

```
# app/controllers/error_controller.rb
class ErrorController
  def handle
    flash.alert(t(".message"))
    redirect_to :back
  end
end
```

举例二：

```
# config/application.rb
config.exceptions_app = self.routes
```

```
# config/routes.rb
match '/404', via: :all, to: 'errors#not_found'
match '/422', via: :all, to: 'errors#unprocessable_entity'
match '/500', via: :all, to: 'errors#server_error'
```

```
# app/controllers/errors_controller.rb
class ErrorsController < ActionController::Base
  layout 'error'

  def not_found
    render status: :not_found
  end

  def unprocessable_entity
    render status: :unprocessable_entity
  end

  def server_error
    render status: :server_error
  end
end
```

```
# app/views

errors/
  not_found.html.erb
  unprocessable_entity.html.erb
  server_error.html.erb
layouts/
  error.html.erb
```

Debug Exceptions

负责在开发环境下，在页面上输出日志和 debug 信息。

有 Show Exceptions 的性质，但不同的是 Show Exceptions 比较通用。而 Debug Exceptions 用于本地开发，因为此时我们希望看到更多、更友好的错误信息。

middleware/templates/ 目录下的文件，直接和它相关。

Request Id

使用现成(请求里面有的话)，或为每个请求生成唯一的 X-Request-Id.

可以在防火墙、负载均衡、Web服务器之间传递，可用于跟踪用户，方便日志处理；或查看请求在集群中哪台机器上处理。

Rails 里可通过 `request.uuid` 查看，相关 HTTP header 是 X-Request-Id 标识。

根据经验，实际项目中可以考虑将此 middleware 移除。

Remote Ip

计算出用户的眞实 IP 地址，可用于防止 IP 伪造等。

相关 HTTP header 是 `X-Forwarded-For (XFF)` 标识。

伪造这一字段比较容易，对于一般的攻击者而言，它管用；但对于高明的攻击者而言，可以考虑将此 middleware 移除。

Reloader

保证在请求之前和响应之后做某事。这里重点是"保证"，具体是什么事，怎么做，不由它决定。

类方法：

```
to_prepare

to_cleanup
```

`to_prepare` 保证在每次请求之前，执行它给的 block.
`to_cleanup` 保证在每次响应之后，执行它给的 block.

实例方法(middleware 的 call 方法会调用它们)：

```
prepare! # 执行 to_prepare 给的 block.

cleanup! # 执行 to_cleanup 给的 block.
```

类方法(作用同上，但我们可以手动调用)：

```
prepare!

cleanup!
```

开发环境下，我们会经常更改代码，但我们又希望请求、响应速度加快。此时，可以使用此 middleware，因为它可以帮我们达到愿望。

Params Parser

解析，并格式化请求过来的参数。

包括但不限于，当请求的数据格式是 JSON 类型时，进行转换：

```
data = ActiveSupport::JSON.decode(request.raw_post)
data = {:_json => data} unless data.is_a?(Hash)

Request::Utils.deep_munge(data).with_indifferent_access
```

注意：不是所有参数，特指 `request.request_parameters` ，也就是 POST 请求发过来的参数。

Flash

提供对 flash 消息的创建、读/写、删除等相关操作。

有 FlashHash 和 FlashNow 两个类。

`flash` 是 FlashHash 的实例对象；
`flash.now` 是 FlashNow 的实例对象。

```
[]
[]=

now

alert
alert=

notice
notice=

empty?

clear

delete

discard

key?
keys

to_hash

each

initialize_copy

keep
```

使用上就能感觉得出 flash 消息兼有 Hash 和 Enumerable 实例对象的部分特性。

至于如何使用，可以查看【Flash 相关使用】章节。

Cookies & ChainedCookieJars

可以通过 ActionController#cookies 读/写 cookies 数据。

基本的写(由 Cookies 提供):

```
# 简单的 cookie 数据
# 浏览器关闭则删除
cookies[:user_name] = "david"

# cookie 存储字符串数据，其它格式要转化。
cookies[:lat_lon] = JSON.generate([47.68, -122.37])

# 设置 cookie 数据的生命周期为一小时
cookies[:login] = { value: "XJ-122", expires: 1.hour.from_now }
```

高级的写(由 ChainedCookieJars 提供)：

```
# cookie 数据签名(用到secrets.secret_key_base)，防止用户篡改。
# 可以使用 cookies.signed[:name] 获取这签名后的数据
cookies.signed[:user_id] = current_user.id

# 设置一个"永久" cookie，默认生命周期是 20 年。
cookies.permanent[:login] = "XJ-122"

# 你可以链式调用以上方法
cookies.permanent.signed[:login] = "XJ-122"
```

读:

```
cookies[:user_name]           # => "david"
cookies.size                  # => 2
JSON.parse(cookies[:lat_lon]) # => [47.68, -122.37]
cookies.signed[:login]        # => "XJ-122"
```

删除:

```
cookies.delete :user_name
```

除上面提到的 signed 和 permanent 外，ChainedCookieJars 还提供方法：

```
# cookie 数据加密(用到 secrets.secret_key_base)，类似于 signed 方法
# 可以使用 cookies.encrypted[:discount] 获取这签名后的数据(已经自动解
密)
cookies.encrypted[:discount] = 45

signed_or_encrypted
```

Session

主要是配置 **Session** 的存储方式。

可选 Cache Store && Cookie Store && Mem Cache Store

用得最多，并且默认的是 CookieStore.

可以在 config/initializers/session_store.rb 配置你的 session 存储方式:

```
Rails.application.config.session_store :cookie_store, key: '_your_app_session'
```

可以在 config/secrets.yml 配置生成 session 的密钥:

```
development:
  secret_key_base: 'secret key'
```

(密钥除签名、加密 session 外，还有其它用途)

运行 `rake secret` . 会生成随机字符串，你可以使用它们做为 secret_key.

> Note: 如果你更改了 secret_key 的话，之前的 session 就不能使用了。因为 secret_key 还有其它用途，一般不建议更改。

Callbacks

可以在某个 middleware 执行之前、之后，执行一些回调方法。

因为我们 middleware 本身就是链式调用，一个个执行，所以这里的回调意义不大。

```
define_callbacks :call
```

任何一个 middleware (或者说 Rack)很重要的两个方法：

```
def initialize(app)
  @app = app
  # ...
end

def call(env)
  # ...
    @app.call(env)
  # ...
end
```

~~Load Interlock~~

主要解决 eager_load 和 cache_classes 都 false 的情况下，仍然能很好的处理并发并且不怎么影响性能。

> 这和移除 Rack::Lock 及 config.threadsafe! 有一定关联。
>
> 简单说：Rack::Lock 多进程的话，它没能解决什么实质问题，还影响性能。 多线程的话，它也不能很好的解决问题，并且还暴露出自身存在的缺陷。

本模块已从 Rails 移除。

Executor

非 Rails 自带的 Middleware，reload 也会影响到。但它们被封装到 ::Rack::BodyProxy 里。

引入此模块的同时，把 Load Interlock 移除了。

Debug Locks

死锁诊断，具体说明及使用方法，可以查看官方文档。

Action Dispatch Http

在一次完整的 **HTTP** 里，提供客户端的 **request** 对象，和服务端的 **response** 对象。

它和 Web 服务器的关系比较近，影响的主要是 http 相关的部分(也就是 request 和 response)，和我们的业务逻辑没有直接关联。

虽然，大部分为 Controller 所用，但要区别开来。Controller 属于 MVC 里的 C，和我们的业务逻辑有着直接关联，而它不是这样的。

Request 和 Response 是连接 ActionController 和 ActionDispatch::Http 主要方式，另外还有很小一部分用其它方式实现。

每一个请求都有 request 和 response.

`request` 为 ActionDispatch::Request 的实例对象。
`response` 为 ActionDispatch::Response 的实例对象。

Request

常用到的 **request**，它还有很多方法在这。

除了提供众多和请求有关的方法外，整个环境也很重要，例如 Middleware 非常需要它们 `ActionDispatch::Request.new(env)`

Request 文件下的内容

对 http://blog.test.example.com:3000/users?name=kelby 发起 GET 请求。

在 new 页面，对 http://blog.test.example.com:3000/users 发起 POST 请求。

查询、请求参数

```
GET & query_parameters
=> {"name"=>"kelby"}
=> {}

POST & request_parameters
=> {}
=> {"utf8"=>"✓",
    "authenticity_token"=>"+xxx.../yyy...==",
    "user"=>{"name"=>"kelby", "email"=>""},
    "commit"=>"Create User"}
```

请求方法

```
form_data?
=> false
=> true

delete?
=> false
=> false

get?
=> true
=> false

head?
=> false
=> false

patch?
=> false
=> false

post?
=> false
=> true

put?
=> false
=> false

xhr? & xml_http_request?
=> false
=> nil

local? # 来自本地请求？用正则判断，所以不是返回 true/false, 而是 0/nil.
=> 0
=> 0
```

一些数据

```
media_type
```

```
=> ""
=> "application/x-www-form-urlencoded"

method
=> "GET"
=> "POST"

method_symbol
=> :get
=> :post

raw_post
=> ""
=> "utf8=%E2%9C%93&authenticity_token=%...%...%3D%3D \n
    &user%5Bname%5D=kelby&user%5Bemail%5D=&commit=Create+User"

request_method
=> "GET"
=> "POST"

request_method_symbol
=> :get
=> :post

server_software
=> "thin"
=> "thin"

uuid
=> "6c1c3684-d8aa-4c03-97d3-d179997773ef"
=> "050daae4-c889-4d24-b627-ba682370216d"

ssl? # 来自于 Rack::Request, 当前是否在是用 https 加密协议。
=> false
=> false

scheme # 来自于 Rack::Request
=> "http"
=> "http"
```

一些对象

```
headers
=> ActionDispatch::Http::Headers 的实例对象(一大串东西)
=> ActionDispatch::Http::Headers 的实例对象(一大串东西)

authorization
=> nil
=> nil

body
=> StringIO 的实例对象
=> StringIO 的实例对象

content_length
=> 0
=> 187

cookie_jar
=> ActionDispatch::Cookies::CookieJar 的实例对象(一大串东西)
=> ActionDispatch::Cookies::CookieJar 的实例对象(一大串东西)

flash
=> ActionDispatch::Flash::FlashHash 的实例对象
=> ActionDispatch::Flash::FlashHash 的实例对象
```

```
check_path_parameters!
=> {:controller=>"users", :action=>"index"}
=> {:controller=>"users", :action=>"create"}
```

客户端路径相关

```
fullpath
=> "/users?name=kelby"
=> "/users"

original_fullpath
=> "/users?name=kelby"
=> "/users"

original_url
=> "http://blog.test.example.com:3000/users?name=kelby"
=> "http://blog.test.example.com:3000/users"
```

服务端 **IP** 相关

```
ip
=> "127.0.0.1"
=> "127.0.0.1"

remote_ip
=> "127.0.0.1"
=> "127.0.0.1"
```

其它方法

```
key?

deep_munge

reset_session
session_options=

parse_query
```

除上述方法外，还有：

```
include ActionDispatch::Http::Cache::Request
include ActionDispatch::Http::MimeNegotiation
include ActionDispatch::Http::Parameters
include ActionDispatch::Http::FilterParameters
include ActionDispatch::Http::URL
```

URL

前面 **request** 里带 **url** 或 **path** 的方法有几个和它相关。

在 new 页面，对 http://blog.test.example.com:3000/users 发起 POST 请求。

检测结果(它们是实例方法)：

```
request.domain
# => "example.com"

request.host
# => "blog.test.example.com"

request.host_with_port # 取得带端口的主机名
# => "blog.test.example.com:3000"

request.optional_port
# => 3000

request.port
# => 3000

request.port_string
# => ":3000"

request.protocol # 取得当前使用网络协议
# => "http://"

request.raw_host_with_port # 代理服务器的主机名和端口
# => "blog.test.example.com:3000"

request.server_port
# => 3000

request.standard_port # 返回网络协议标准端口(http 为 80, https 为 44
3)
# => 80
```

```
request.standard_port? # 判断当前协议是否是标准端口
# => false

request.subdomain
=> "blog.test"
request.subdomain 2
# => "blog"

request.subdomains
# => ["blog", "test"]
request.subdomains 2
# => ["blog"]

request.url # 取得当前 requset 完整 url
# => "http://blog.test.example.com:3000/users"
```

除上述方法外，还有(模块方法)：

```
extract_domain
extract_subdomain
extract_subdomains

full_url_for
path_for
url_for
```

以 ActionDispatch::Http::URL.x 的形式调用它们。

数据来源

```
env["HTTP_X_FORWARDED_HOST"]
env['HTTP_HOST'], env['SERVER_NAME'], env['SERVER_ADDR'], env['SERVER_PORT']

以及父类 Rack::Request (它也是从 env 里取数据，然后处理。具体不在此描述)
```

Headers

通过它可以访问当前环境下 HTTP 请求头(将 HTTP headers 转换成环境变量)

数据结构和 Hash 类似，所以提供的方法也类似，这里就不列举了。

使用举例：

```
env     = { "CONTENT_TYPE" => "text/plain" }
headers = ActionDispatch::Http::Headers.new(env)
headers["Content-Type"] # => "text/plain"
```

对应 Rails 里的 `request.headers` 而不是 `headers`

前者，是这里 ActionDispatch::Http::Headers 的实例对象； 而后者，是 ActionDispatch::Response.default_headers 为 Hash 实例对象（特指 Http Response 里的 Headers）

通过它，可获取很多 CGI_VARIABLES 相关值。

> Note: 它包含了很多内容(少量 env, 一般 rack, 很多 action_dispatch, 少量 action_controller)，感兴趣可以看看。

Mime Negotiation

在 new 页面，对 http://blog.test.example.com:3000/users 发起 POST 请求。

检测结果：

```
request.accepts
=> [#<Mime::Type:0x007f8a1a498cd8 @string="text/html", @symbol=:
html, \n
                                  @synonyms=["application/xhtml+
xml"]>,
 #<Mime::Type:0x007f8a20f2f498 @string="image/webp", @symbol=nil
, @synonyms=[]>,
 #<Mime::Type:0x007f8a1a48a908 @string="application/xml", @symbo
l=:xml, \n
                               @synonyms=["text/xml", "applicati
on/x-xml"]>,
 #<Mime::Type:0x007f8a20f2f3f8 @string="*/*", @symbol=nil, @syno
nyms=[]>]

request.content_mime_type
=> #<Mime::Type:0x007f8a1a47be58 @string="application/x-www-form
-urlencoded", \n
                                 @symbol=:url_encoded_form, @syn
onyms=[]>

request.content_type
=> "application/x-www-form-urlencoded"

# = formats.first
request.format
=> #<Mime::Type:0x007f8a1a498cd8 @string="text/html", @symbol=:h
tml, \n
                                 @synonyms=["application/xhtml+x
ml"]>

request.formats
=> [#<Mime::Type:0x007f8a1a498cd8 @string="text/html", @symbol=:
html, \n
                                  @synonyms=["application/xhtml+
xml"]>]
```

除上述方法外，还有：

```
negotiate_mime

format=
formats=
variant=
```

给“变种”增加了一些新的方法：

```
request.variant = :phone
request.variant.phone?  # true
request.variant.tablet? # false

request.variant = [:phone, :tablet]
request.variant.phone?                  # true
request.variant.desktop?                # false
request.variant.any?(:phone, :desktop)  # true
request.variant.any?(:desktop, :watch)  # false
```

Parameters

常用到的 **params (也叫 parameters)**等。

```
parameters & params

path_parameters
```

在 new 页面，对 http://blog.test.example.com:3000/users 发起 POST 请求。

检测结果：

```
request.parameters
=> {"utf8"=>"✓",
 "authenticity_token"=>"+.../hHj0C5Ao7CxXYwlejyghpNM3elUVw==",
 "user"=>{"name"=>"kelby", "email"=>""},
 "commit"=>"Create User",
 "controller"=>"users",
 "action"=>"create"}

# 另：
request.parameters == params
=> true
# 但它们两者并不是完全等价的

request.path_parameters
=> {:controller=>"users", :action=>"create"}
```

Parameter Filter & Filter Parameters

过滤参数。如：

```
env["action_dispatch.parameter_filter"] = [:password]
# => replaces the value to all keys matching /password/i with "[
FILTERED]"
```

这就是为什么我们在日志里看不到 password 的值，而是显示 FILTERED 的原因。

实例方法：

```
filter

# 具体包含下面 3 项内容：

filtered_parameters
filtered_env
filtered_path
```

在 new 页面，对 http://blog.test.example.com:3000/users 发起 POST 请求。

检测结果：

```
filtered_env
# => 一大串 env 相关数据

request.filtered_parameters
# => {"utf8"=>"✓",
 "authenticity_token"=>"+.../hHj0C5Ao7CxXYwlejyghpNM3elUVw==",
 "user"=>{"name"=>"kelby", "email"=>""},
 "commit"=>"Create User",
 "controller"=>"users",
 "action"=>"create"}

request.filtered_path
# => "/users"
```

使用举例：

```
# 用 "[FILTERED]" 替换 /password/i
env["action_dispatch.parameter_filter"] = [:password]

# 用 "[FILTERED]" 替换 /foo|bar/i
env["action_dispatch.parameter_filter"] = [:foo, "bar"]
```

以上直接使用 env 比较少见，更多的用法是：

```
# filter_parameter_logging.rb
Rails.application.config.filter_parameters += [:password]
```

查看应用程序过滤了什么字段：

```
Rails.application.config.filter_parameters
```

除上述方法外，还有：

```
env_filter

filtered_query_string

parameter_filter_for

parameter_filter
```

> Note: 用得比较多的是方法 parameter_filter，通常用来配置过滤 :password ... 过滤的其实是 ParameterFilter 的实例对象，虽然这里没有体现出来。

Cache 之 Request

HTTP Cache 里的 **Request** 部分。如：检测 ETag、Cache-Control

一般来说，客户端(通常是浏览器)也有缓存机制，我们可以设置 ETag 和 Last-Modified 告诉它们资源是否已更新，缓存是否已过期。

一般来说，我们不会手动调用这里的方法。

提供方法：

```
etag_matches?

fresh?

if_modified_since
if_none_match
if_none_match_etags

not_modified?
```

~~Uploaded File~~

上传文件时用到，默认文件放到 :tempfile 里，相关会用到它来处理。

```
close
open
read
rewind

path
size

eof?
```

在 Http Parameters 和 Metal Parameters 里有用到它，对应 params 里有 :tempfile 和 params.permit(:x) 里包含文件对象时处理。

使用举例：

```
Factory.define :image do |o|
  file = File.new(Rails.root + 'spec/fixtures/images/rails.png')
  file.rewind

  o.file ActionDispatch::Http::UploadedFile.new(:tempfile => fil
e,
                                                :filename => Fil
e.basename(file))
end

@image = Factory(:image)
```

Utils

执行 GET 或 POST 请求时，对参数进行检测，不是 UTF-8 可能会报错。但，实际上它起的作用很小。

Response

常用到的 **response**，它还有很多方法在这。

Response 文件下的内容

和 ENV 有关的方法：

```
auth_type
gateway_interface
path_translated

remote_host
remote_ident
remote_user
remote_addr

server_name
server_protocol

accept
accept_charset
accept_encoding
accept_language

cache_control
from
negotiate
pragma
```

其它：

```
abort

await_commit
await_sent

body
body=
body_parts
```

```
close

code

commit!

committed?

content_type=

cookies

delete_cookie

location & redirect_url
location=

message & status_message

response_code

sending!
sending?

sent!
sent?

set_cookie

status=

to_a & prepare!
to_ary
```

~~Filter Redirect~~

```
filtered_location
```

配置 config.filter_redirect，可以从日志里过滤掉一些敏感的"重定向地址"：

```
config.filter_redirect << 'www.rubyonrails.org'
```

可以使用字符串、正则表达式，或者一个数组(包含字符串或正则表达式)：

```
config.filter_redirect.concat ['www.rubyonrails.org', /private_p
ath/]
```

匹配的 URL 会显示为 '[FILTERED]'。

查看应用程序过滤了什么字段：

```
Rails.application.config.filter_redirect
```

Cache 之 Response

HTTP Cache 里的 **Response** 部分。如：设置 ETag、Cache-Control.

一般来说，客户端(通常是浏览器)也有缓存机制，我们可以设置 ETag 和 Last-Modified 告诉它们资源是否已更新，缓存是否已过期。

一般来说，我们不会手动调用这里的方法。

提供方法：

```
date
date=
date?

etag=
weak_etag=
strong_etag=
etag?
weak_etag?
strong_etag?

last_modified
last_modified=
last_modified?
```

其它

TODO

Rack Cache

Rack::Cache 也属于 Rack middleware，可用后端服务存储 assets(静态资源) 内容，默认放在 public/ 目录下。

现在默认没有采用：

```
config.action_dispatch.rack_cache = false
```

你可以配置，如：

```
Rails.configuration.action_dispatch.rack_cache
# => {:metastore=>"rails:/", :entitystore=>"rails:/", :verbose=>
false}
```

Rails 对应有 Rails Meta Store 和 Rails Entity Store 类，它们都有：

```
def initialize(store = Rails.cache)
  @store = store
end
```

并且都有对 @store 的 read、write 等方法。

链接 rack-cache

Mime Type register

设置响应类型 **Mime Type.**

先看看 `register` 方法

第 1，2 个参数容易理解
第 3 个为标准类型
第 4 个为扩展类型

使用举例：

```
Mime::Type.register "text/html", :html, %w( application/xhtml+xm
l ), %w( xhtml )
```

`register` 只是登记，本身是没有处理能力的，还需要使用 Action Controller 添加渲染器并处理：

```
ActionController::Renderers.add ...
```

查看 Rails 支持什么类型的 MIME(Multipurpose Internet Mail Extensions，多用途互联网邮件扩展)：

```
Mime::SET

=> [#<Mime::Type:0x007fdf7f01d9f0
  @string="text/html",
  @symbol=:html,
  @synonyms=["application/xhtml+xml"]>,
 #<Mime::Type:0x007fdf7f01d6f8
  @string="text/plain",
  @symbol=:text,
  @synonyms=[]>,
 #<Mime::Type:0x007fdf7f01d400
  @string="text/javascript",
  @symbol=:js,
  @synonyms=["application/javascript", "application/x-javascript
"]>,
 #<Mime::Type:0x007fdf7f01d158 @string="text/css", @symbol=:css,
 @synonyms=[]>,
 ... ...
```

~~Session~~

在文件目录上很有意思，它不是放在 http/ 而是放在单独的 request/ 目录下。

这是因为和其它 http/ 下的模块相比，它不仅被 request 调用，同时它在 middleware Flash 里也被调用。

Action Dispatch RouteSet 底层

Routing 定义应用的路由后，由 RouteSet 底层实现。

本章节偏低层，如果你不能掌握，可以先跳过。

RouteSet 概述

- 特指 route_set.rb 及 routes_proxy.rb 两文件
- 本身就充满魔法，是 Routing 里的一个模块。
- 还是内外沟通的桥梁。
- 内指 Journey.
- 外指对外的接口及 routing 目录里的其它内容。

```
require 'action_dispatch'

routes = ActionDispatch::Routing::RouteSet.new

routes.draw do
  get '/' => 'mainpage#index'
  get '/page/:id' => 'mainpage#show'
end
```

从 Action Dispatch 转换站场到 Action Controller. (准确点：Action Dispatch -> Metal -> Abstract Controller -> Action Controller)

```
# route_set.rb
def dispatch(controller, action, env)
  controller.action(action).call(env)
end
```

除上述外，还有：

```
Routes Proxy # 从 RouteSet 里抽取而来
```

实例对象和各个实例方法

实例对象

在 Rails 源代码里， `app.routes` 指的是 RouteSet 的实例对象。

```
# action_mailer/railtie.rb
extend ::AbstractController::Railties::RoutesHelpers.with(app.ro
utes, false)
include app.routes.mounted_helpers

# action_controller/railtie.rb
include app.routes.mounted_helpers
extend ::AbstractController::Railties::RoutesHelpers.with(app.ro
utes)

# action_dispatch/routing/mapper.rb
app.routes.define_mounted_helper(name)
app.routes.extend Module.new { ... }

# rails/application/finisher.rb
app.routes.append do ... end
app.routes.define_mounted_helper(:main_app)
```

在 Rails 源代码里， `@set` 有时候也是 RouteSet 的实例对象，如 Mapper 里：

```
# action_dispatch/routing/mapper.rb
initialize
  @set = set
  @scope = Scope.new({ :path_names => @set.resources_path_names
})

dispatcher
  @set.dispatcher defaults

default_url_options=
  @set.default_url_options = options

has_named_route?
  @set.named_routes.routes[name.to_sym]

define_generate_prefix
  _route = @set.named_routes.get name
  _routes = @set

add_route
  mapping = Mapping.build(@scope, @set, URI.parser.escape(path),
 as, options)
  @set.add_route(app, conditions, requirements, defaults, as, an
chor)

name_for_action
  candidate unless candidate !~ /\A[_a-z]/i || @set.named_routes
.key?(candidate)
```

也正因为如此，除了 `draw` 和 `add_routes` 外，我们可以在代码里搜索，看哪里的代码影响了路由规则的生成。

```
# route_set.rb
attr_accessor :formatter, :set, :named_routes, :default_scope, :
router
attr_accessor :disable_clear_and_finalize, :resources_path_names
attr_accessor :default_url_options, :request_class

alias :routes :set

# formatter 是 Journey::Formatter 的实例对象
# set 是 Journey::Routes 的实例对象
# named_routes 是 NamedRouteCollection 的实例对象
# router 是 Journey::Router 的实例对象
# resources_path_names 也就是 default_resources_path_names，默认有
：new 和 edit
# request_class 默认是 ActionDispatch::Request
```

各个实例方法

```
inspect & to_s
routes & set

draw

# 往变量 @append 和 @prepend 里塞东西
append
prepend

finalize!

# 消除 named_routes、set、formatter
clear!

dispatcher

# 包含了所有 Engine 提供的 helper 方法，和 main_app 方法。
mounted_helpers

# 新增 Engine 提供的 helper 方法。
```

```
# 用到了 MountedHelpers 和 RoutesProxy
define_mounted_helper

# 它其它是一个 Module
# 所以我们可以 include Rails.application.routes.url_helpers
# 包含了所有和 URL 有关的 helper 方法(也就是所有 x_url 和 x_path 方法)。

url_helpers

# 没有路由规则？
empty?

# 会调用到以下方法：
# build_path
# build_conditions
# @set.add_route
# named_routes[name]
add_route

extra_keys
generate_extras

# generate_extras 和 url_for 调用到
generate

# 我们在路由规则是可以带默认参数的，这在定义的时候就可以知道。
# 这里的意思其实是：有默认参数吗？
optimize_routes_generation?

find_script_name # url_for 调用到

# 可谓是其它组件 url_for 方法的源头
# (当然，得用到路由方法才行)
url_for
path_for # 简单封装 url_for

# 浏览器里输入的网址，最先经过它解析
# 关键部分是调用 @router.recognize
recognize_path
```

`recognize_path` 使用举例：

```
Rails.application.routes.recognize_path "http://localhost:3000/users/1"
# => {:controller=>"users", :action=>"show", :id=>"1"}
```

通过这一步，外部 URL 已经被转换成了 Rails 能够识别的语言。在这之后，Rails 想怎么处理就怎么处理。

```
# 使用 Journey 处理我们提供的路由规则，得到 path 这部分
build_path
```

`draw` 直接对外暴露的接口，初始化 Mapper 对象。然后交还各个模块处理。 本质是：运用 Mapper，处理 config/routes.rb 里的代码。

相关代码：

```
mapper = Mapper.new(self)

if default_scope
  mapper.with_default_scope(default_scope, &block)
else
  mapper.instance_exec(&block)
end
```

之后，就是执行【路由常用方法汇总】章节里提到的方法。

`initialize` 用到了 NamedRouteCollection， 内容是：

```
self.named_routes = NamedRouteCollection.new
```

初始化了几个有意义的变量，如：named_routes、@set、@router、@formatter

`generate` 用到了 Generator，内容是：

```
Generator.new(route_key, options, recall, self).generate
```

`dispatcher` 用到了 Dispatcher 内容是：

```
Routing::RouteSet::Dispatcher.new(defaults)
```

~~Named Route Collection~~

向 **Rails** 其它组件提供路由相的关 **x_path, x_url** 方法。

include Enumerable，所以看到很多同名方法也就不奇怪了。它们意义和使用方式雷同，不再一一解释。

```
module ActionDispatch
  module Routing
    class RouteSet
      def initialize(request_class = ActionDispatch::Request)
        self.named_routes = NamedRouteCollection.new
        # ...
      end
    end
  end
end
```

各个实例方法

```
routes

url_helpers_module

route_defined?

helper_names

clear! & clear

add & []= # 主要作用，添加 x_url 和 x_path 辅助方法
get & []

names

path_helpers_module
```

除上述方法外，还有：

```
key?

each

length
```

另外，还有私有方法 `define_url_helper`
我们使用的路由相关的 x_url 和 x_path 辅助方法就是由它而来的。

```
def define_url_helper(mod, route, name, opts, route_key, url_strategy)
  helper = UrlHelper.create(route, opts, route_key, url_strategy)
  mod.module_eval do
    define_method(name) do |*args|
      # ...
    end
  end
end
```

~~Url Helper~~

非常底层的实现，主要对外接口 self.create

当调用 Named Route Collection 的私有方法 define_url_helper 时会用到，部分内容是：

```
helper = UrlHelper.create(route, opts, route_key, url_strategy)
```

哪里调用它了？

1) `@set.add_route` 添加路由规则时有调用 named_routes[name]

2) 而 `named_routes` 就是 NamedRouteCollection 的实例对象：

```
attr_accessor :named_routes

def initialize
  self.named_routes = NamedRouteCollection.new
  # ...
end
```

3) 并且 `[]=` 方法实际就是 add 方法，而 add 又调用了上面的 define_url_helper

~~Dispatcher~~

将 **HTTP** 请求向具体的 **Controller#action** 转移。

继承于 Routing::Endpoint

```
module ActionDispatch
  module Routing
    class RouteSet
      def dispatcher(defaults)
        Routing::RouteSet::Dispatcher.new(defaults)
      end
    end
  end
end
```

实例方法：

```
dispatcher?

serve

prepare_params!

controller
```

私有方法：

```
dispatch

controller_reference

normalize_controller!
merge_default_action!
```

最重要的方法是 `dispatch`，将战场切换到 Controller#action

```
def dispatch(controller, action, env)
  controller.action(action).call(env)
end
```

每一条路由规则，对应着一个 **Dispatcher** 实例。

每一个路由规则转换着 `draw` 对应一个 Dispatcher 实例。

用最简单的 get 方法举例：

```
AppName::Application.routes.draw do
  get 'photos/:id' => 'photos#show', :defaults => { :format => '
jpg' }
end
```

生成实例：

```
app: #<ActionDispatch::Routing::RouteSet::Dispatcher:0x007fd05e0
cf7e8
           @defaults={:format=>"jpg", :controller=>"photos", :ac
tion=>"show"},
           @glob_param=nil,
           @controller_class_names=#<ThreadSafe::Cache:0x007fd05
e0cf7c0
           @backend={},
           @default_proc=nil>>
conditions: {:path_info => "/photos/:id(.:format)",
             :required_defaults => [:controller, :action],
             :request_method => ["GET"]}
requirements: {}
defaults: {:format=>"jpg", :controller=>"photos", :action=>"show"
}
as: nil
anchor: true
```

`app` 如果条件匹配，将要执行的 Rack application.

`conditions` 匹配条件。执行此 Rack application 要匹配什么条件。

`defaults` 默认参数。

`requirements` 对应方法里的 :constraints 参数。

`as` 对应方法里的 :as 参数。

~~Generator~~

url_for 方法的重要组成部分。

```
module ActionDispatch
  module Routing
    class RouteSet
      def generate(route_key, options, recall = {})
        Generator.new(route_key, options, recall, self).generate
      end
    end
  end
end
```

生成失败的话，会报 UrlGenerationError 错误。

各个实例方法：

```
generate

# 以下几个方法 initialize 时被调用
normalize_recall!
normalize_options!
normalize_controller_action_id!
use_relative_controller!
normalize_controller!
normalize_action!
```

initialize 和 `generate` 对外提供的接口。

除上述外，还有：

```
use_recall_for

controller
current_controller

different_controller?
```

和

```
attr_reader :options, :recall, :set, :named_route
```

~~Routes Proxy~~

Rails 项目，默认使用 `main_app` 而使用 gem 'rails_admin' 的话，会有 `rails_admin` 相关路由。它们都是 RoutesProxy 的实例。

```
module ActionDispatch
  module Routing
    class RouteSet
      def define_mounted_helper(name)
        # ...

        routes = self
        MountedHelpers.class_eval do
          define_method "_#{name}" do
            RoutesProxy.new(routes, _routes_context)
          end
        end

        # ...
      end
    end
  end
end
```

```
mount SomeRackApp, at: "some_route"
```

如果 mount 的 Engine 恰好也是一个 Rails app，那么就会生成带前缀的路由 helper 方法。

此时，Engine 内外，Engine 之间如何通信？通过代理(也就是 Routes Proxy 的实例对象)。

~~Journey~~

路由生成、解析底层实现

重要组成部分有：

- Routes
- Router
- Formatter

RouteSet 里调用到它的几个方法：

initialize

```
module ActionDispatch
  module Routing
    class RouteSet
      attr_accessor :formatter, :set, :named_routes, :default_sc
ope, :router
      attr_accessor :disable_clear_and_finalize, :resources_path
_names
      attr_accessor :default_url_options, :request_class

      alias :routes :set

      def initialize(request_class = ActionDispatch::Request)
        self.named_routes         = NamedRouteCollection.new
        self.resources_path_names = self.class.default_resources
_path_names
        self.default_url_options  = {}
        self.request_class        = request_class

        @append                     = []
        @prepend                    = []
        @disable_clear_and_finalize = false
        @finalized                  = false

        @set       = Journey::Routes.new
        @router    = Journey::Router.new @set
        @formatter = Journey::Formatter.new @set
      end
    end
  end
end
```

build_path

```
module ActionDispatch
  module Routing
    class RouteSet
      def build_path(path, ast, requirements, anchor)
        strexp = Journey::Router::Strexp.new(
            ast,
            path,
            requirements,
            SEPARATORS,
            anchor)

        pattern = Journey::Path::Pattern.new(strexp)

        builder = Journey::GTG::Builder.new pattern.spec

        # Get all the symbol nodes followed by literals that are
 not the
        # dummy node.
        symbols = pattern.spec.grep(Journey::Nodes::Symbol).find
_all { |n|
          builder.followpos(n).first.literal?
        }

        # ... ...

        pattern
      end
    end
  end
end
```

call

```
module ActionDispatch
  module Routing
    class RouteSet
      def call(env)
        req = request_class.new(env)
        req.path_info = Journey::Router::Utils.normalize_path(re
q.path_info)
        @router.serve(req)
      end
    end
  end
end
```

recognize_path

```
module ActionDispatch
  module Routing
    class RouteSet
      def recognize_path(path, environment = {})
        method = (environment[:method] || "GET").to_s.upcase
        path = Journey::Router::Utils.normalize_path(path) unless
 path =~ %r{://}
        extras = environment[:extras] || {}

        env = Rack::MockRequest.env_for(path, {:method => method
})

        req = request_class.new(env)
        @router.recognize(req) do |route, params|
          # ... ...
        end
      end
    end
  end
end
```

路由已经定义，请求过来如何匹配？

之前用的是正则匹配，例如 "/pictures/A12345" 可以匹配到：

```
get 'pictures/:id' => 'pictures#show', :constraints => { :id =>
/[A-Z]\d{5}/ }
```

引入 journey 后，性能及其它各方面都有了很大的改善。

报错 ActionController::UrlGenerationError

当 Formatter 尝试使用 `generate` 将 `params` 传递过来的参数解析，但是解析不了的时候，就会报此错误。

Action Cable

TODO

服务端 - **Ruby** 代码

TODO

Channel

可直接调用。类似 Controller ... 外部请求过来，MVC 里最先由它处理。

它处理从客户端过来的 ws 请求。

Base

完成请求转发，由 Action Dispatch 到具体的 Channel#action.
身份地位相当于 Action Controller 里的 Metal + Abstract Controller 里的 Base.

另外，它是 Channel 的头：

```
include Callbacks
include PeriodicTimers
include Streams
include Naming
include Broadcasting
```

内部处理

Channel 整体结构比 Controller 简单，这里的 Base 就相当于 Action Controller 里的 Metal.

它有类方法 `action_methods` 、实例方法 `perform_action` 和私有方法 `dispatch_action` 等。Action Dispatch 转发请求过来后，主要由它进行处理，转给具体的 Channel#action.

区别也是有的，这边的请求不需要 Middleware 进行处理，所以它没有调用到，而 Metal 需要。

对外提供接口

类方法：

```
action_methods
```

```
method_added

clear_action_methods!
```

实例方法：

```
attr_reader :params, :connection, :identifier
delegate :logger, to: :connection
```

```
perform_action

unsubscribe_from_channel
```

`perform_action` public_send 后执行我们自定义的 action.

```
# 实际上由 connection 完成
transmit

# 以下两个由子类实现
subscribed
unsubscribed

# 以下几个相当于配置、问询

reject

defer_subscription_confirmation?
defer_subscription_confirmation!

subscription_confirmation_sent?
subscription_rejected?
```

默认其子类要重写 `subscribed` 和 `unsubscribed` 方法。

`subscribed` 相当于初始化。

如果不允许订阅，可以用 `reject` 进行拒绝。

`transmit` 它只是傀儡，最终交给 ActionCable.connection 执行。

在这里，`reject` 和 `transmit` 就像 Controller 里 `render` 和 `redirect_to` 一样的东西。

Streams

```
# pubsub.subscribe
stream_from

# pubsub.unsubscribe
stop_all_streams

# 封装 stream_from
stream_for
```

> Note：由 `pubsub` 完成实际操作。

```
class CommentsChannel < ApplicationCable::Channel
  def follow(data)
    stream_from "comments_for_#{data['recording_id']}"
  end

  def unfollow
    stop_all_streams
  end
end

CommentsChannel.broadcast_to(@post, @comment)
```

Connection 终于有了自己的一点处理能力，虽然小，但你可以选择使用，也可放弃。

常用 `stream_from` 以处理一对一消息；`stop_all_streams` 单方面即可中止服务。`stream_for` 和 `stream_from` 类似，只是多了一个空间。

```
stream_from -> connection.transmit -> Connection#transmit -> Web
Socket#transmit
```

另：

```
@websocket = ::WebSocket::Driver.websocket?(env) ? ClientSocket.
new(env, event_target, stream_event_loop) : nil
```

Channel 补充

Broadcasting

类方法：

```
broadcast_to(model, message)

broadcasting_for
```

它只是傀儡，最终交给 `ActionCable.server` 执行。

Callbacks

相关回调操作：

```
after_subscribe & on_subscribe
after_unsubscribe & on_unsubscribe

before_subscribe
before_unsubscribe
```

Naming

```
channel_name
```

通过此方法，可以获取通道名字：

```
ChatChannel.channel_name # => 'chat'
Chats::AppearancesChannel.channel_name # => 'chats:appearances'
```

Periodic Timers

```
periodically(callback, every:)
```

可以在 Channel 里使用的类方法，周期性的执行某个指定的回调或 action.

算是一个配置，也不是真正执行。

Connection

Base

这里的"连接"几乎等价于 Websocket 连接。

```
# 简单封装 transmit
beat

close

# 不要直接调用，应由 connect、disconnect 触发
process

receive

send_async

statistics

# 不要直接调用，应由 Channel 实例对象的 transmit 触发
transmit
```

其子类里的 public 实例方法，可直接调用。

注意：可以重写（覆盖）父类的方法，但区别于 public 实例方法，不能直接调用。

```
class AppearanceChannel < ApplicationCable::Channel
  def subscribed
    @connection_token = generate_connection_token
  end

  def unsubscribed
    current_user.disappear @connection_token
  end

  def appear(data)
    current_user.appear @connection_token, on: data['appearing_o
n']
  end

  def away
    current_user.away @connection_token
  end

  private
    def generate_connection_token
      SecureRandom.hex(36)
    end
end
```

示例代码中， `subscribed` 和 `unsubscribed` 由父类定义，不能直接调用。 `generate_connection_token` 私有方法，不能直接调用。 `appear` 和 `away` 可以直接调用。

其它方法：

```
attr_reader :server, :env, :subscriptions, :logger, :worker_pool
delegate :event_loop, :pubsub, to: :server
```

例如，使用 `logger`

```
logger.add_tags current_user.name
```

另外，它提供的一些 protected 方法也很实用：

```
request

cookies
```

这里的 `request` 是 ActionDispatch::Request 的实例对象，所以通过它也可以获取到一些和请求有关的数据。

Identification

类方法：

```
identified_by
```

常用 `identified_by` 进行身份验证。

实例方法：

```
connection_identifier
```

Authorization

```
reject_unauthorized_connection
```

常用 `reject_unauthorized_connection` 进行报错，只能被动地等着被调用。

Connection 补充

WebSocket

```
alive?()

possible?()

transmit(data)
```

Tagged Logger Proxy

```
add_tags(*tags)

tag(logger)

debug
info
warn
error
fatal
unknown
```

```
log(type, message)
```

~~Internal Channel~~

表示带有同一 identifier 的 connection，本质还是 Channel.

~~Message Buffer~~

```
append(message)

process!()

processing?()
```

~~Subscriptions~~

```
add(data)

execute_command(data)

identifiers()

perform_action(data)

remove(data)

remove_subscription(subscription)

unsubscribe_from_all()
```

~~Client Socket~~

~~Stream~~

~~Stream Event Loop~~

~~Faye Client Socket~~

~~Faye Event Loop~~

~~Subscriptions~~

Server

Rails 默认使用的实例是 `ActionCable.server` 除了 Base 提供的实例方法外，它还可调用 Broadcasting 和 Connections 提供的方法。

服务端向客户端发送 ws 请求。

Base

实例对象有：`ActionCable.server`

实例方法：

```
# 所有 Channel 类
channel_classes

# identified_by 指定的标识符
connection_identifiers

# streams/broadcasting 所使用的适配器
# ActionCable::SubscriptionAdapter::Async 实例
pubsub

# ActionCable::RemoteConnections 实例
remote_connections

# ActionCable::Connection::StreamEventLoop 实例
stream_event_loop

# ActionCable::Server::Worker 实例
worker_pool

# 断开带有某标识符的连接
disconnect
```

```
# ActionCable::Server::Configuration 实例
config
```

类方法：

```
logger
```

Broadcasting

发送消息给指定 channel 的订阅者。

实例方法：

```
broadcast

broadcaster_for
```

常用 `broadcast` 进行广播。

```
ActionCable.server.broadcast "web_notifications_1",
  { title: 'New things!', body: 'All shit fit for print' }
```

实际上广播任务由 server 对应的 `pubsub` 来完成。

广播与接收一般要对应

ActionCable.server 用于广播，它包括标识及数据。

ApplicationCable::Channel 用于接收数据并处理，特定的 Channel 一般只接收有特定标识的 server 发过来的请求。

Server 补充

Connections

手动添加、移除链接，主要为了方便开发、调试，建议请不要做它用。

```
connections

add_connection
remove_connection

setup_heartbeat_timer

open_connections_statistics
```

Configuration

实例对象： `ActionCable.server.config`

```
attr_accessor :logger, :log_tags
attr_accessor :connection_class, :worker_pool_size
attr_accessor :redis, :channels_path
attr_accessor :disable_request_forgery_protection, :allowed_requ
est_origins
attr_accessor :url
```

```
channel_paths

channel_class_names

# 默认使用的适配器是 redis
pubsub_adapter
```

Worker

```
Server.send_async
```

Active Record Connection Management

及时清理 Active Record 的链接，防止堆积成山。

Remote Connections

针对使用了 `identified_by` 的连接，方便对其进行批量处理，它并不是什么特殊概念。

第一部分：

```
where
```

```
module ApplicationCable
  class Connection < ActionCable::Connection::Base
    identified_by :current_user
    # ...
  end
end

# 查询、断开连接
ActionCable.server.remote_connections.where(current_user: User.find(1)).disconnect
```

第二部分：

```
disconnect
```

```
identifiers
```

它们只是调用，实际工作由 `server` 完成。

Action Cable Helper

提供 `action_cable_meta_tag` 方法，用于创建 action-cable-url，保存 url.

在 ActionCable 的 Consumer 创建 WebSocket 链接时需要它 url.
当然了，这是自动完成的。

Subscription Adapter

用于适配 ActionCable 的存储方案，默认是 Redis.

具体的存储方案，只要实现下列几个方法即可：

```
broadcast(channel, payload)

subscribe(channel, message_callback, success_callback = nil)
unsubscribe(channel, message_callback)

shutdown
```

在 Server#pubsub 时会用到它来临时存储数据。

Subscription Adapter 补充

目前有 Async、EventedRedis、Inline、PostgreSQL、Redis、SubscriberMap 等 6 种适配。

客户端 - Coffee 脚本

我们用脚本生成 Channel 时，一般地都有对应的 Coffee 脚本。

Consumer 开动机器，Subscription 接受命令，Connection 干活。

Rails 默认使用 `App.cable` 表示其 Consumer 实例。

4 种调用

- 约定调用的 coffee 方法
- 外部直接调用 coffee 方法
- 内部调用自己的方法
- 调用 Channel 里的 action

Action Cable 文件下的内容

根据 action-cable-url 内容，得到 wss 协议的链接，然后构造 Consumer.

```
getConfig
createWebSocketURL
createConsumer
```

其它：

```
startDebugging
stopDebugging
```

Consumer

创建各个相关的实例对象，并间接具有发送 @webSocket 消息的能力。

创建各个相关的实例对象。

由 Consumer 根据 action-cable-url 得到 url (使用协议 ws)，创建 WebSocket 实例。

```
constructor
```

一并创建 Subscriptions、Connection 及 Connection Monitor 实例。

并间接具有发送 **@webSocket** 消息的能力。

```
send
```

想通过 `@webSocket` 发送消息，可以通过 Consumer 实例完成(尽管它是封装的 Connection 对象)。

Subscription

传递消息，以及真正干活的好手。

传递消息到 **Channel#action**

常用的：

```
# 封装下面的 send
perform
```

其它：

```
# 实质是 consumer.send -> connection.send
send
```

```
unsubscribe
```

创建 subscription 需要 channel.

常用方法 `perform` 可以在 Coffee 里调用 Channel 里的 action.

```
Subscription#perform -> Subscription#send -> Consumer#send -> Co
nnection#send -> WebSocket#send
```

回调响应，及真正干活

我们可以(和普通 JS 一样)在它的基础上自定义方法，进行调用。

Connection

表面操作的是 connect，实质是直接操作 websocket ...

```
send

open
close
reopen

disconnect
```

它所打开、连接的就是 WebSocket 连接。

```
isOpen
isState

getState

installEventHandlers

events
```

不过我们很少直接使用它。

客户端需要发消息给服务端的话，可以让 Subscription 调用，然后通过 Consumer 中转，最后才由 Connection 进行发送。

Subscriptions

Subscription 的集结地，有专门的 `@subscriptions` 用来保存 subscription，并且可进行管理。

常用的：

```
create
```

通过它可创建 Subscription 实例。

其它：

```
constructor
```

```
add
remove
reject
forget
reload
notify

findAll
notifyAll

sendCommand
```

还有各种方法，针对具体某个 subscription 进行各种操作。可订阅某个 Channel，之后在其 Channel 发布消息时，它就能收到了。

通过 identifier 查找具体某个 subscription.

Connection Monitor

轮询机制，通过浏览器自带特性就能实现，非常重要的“约定”。

配置：

```
@pollInterval:
  min: 3
  max: 30

@staleThreshold: 6
```

常用方法如：

```
connected
disconnected

received
```

其它：

```
identifier
constructor

reset
start
stop
poll

getInterval

reconnectIfStale

connectionIsStale
disconnectedRecently

visibilityDidChange

now

secondsSince

clamp
```

也是通过“死循环”实现的。
这里的“死循环”是有间隔、有状态、可控的。

一些重要的东西

ActionCable.server

Rails 默认使用的实例是 `ActionCable.server` 除了 Base 提供的实例方法外，它还可调用 Broadcasting 和 Connections 提供的方法。

App.cable

Rails 默认使用 `App.cable` 表示其 Consumer 实例。

```
@App = {}
App.cable = ActionCable.createConsumer()
```

术语

server 服务端（死的）

connection 链接本身（活的）

consumer 客户端（死的）

channels 链接通道（死的）

broadcastings 对通道的某些操作（活的）

流程

1）A输入数据，jQuery 监测，发送类似 Ajax 请求；服务器收到，服务器处理后，发送 ws 请求 1）或由服务端 server.broadcast 直接发送 ws 请求

发送的 ws 请求，要有标识及数据。

2）客户端始终在等待 ws 请求 3）客户端收到 ws 请求，根据标识由对应的 Channel js 处理

此处有分支

4）客户端直接使用 JQuery 处理，不需要再请求服务器 4）客户端 @perform 发送 ws 请求，对应 Channel rb 里的 action 处理，再响应...server broadcast 或 Streams；

结束。

重要方法：客户端 perform, 服务端 broadcast 和 received.

服务端、客户端均可发送 ws 请求；简单的只有一次 ws 请求，复杂一点的有两次 ws 请求。

Action Cable others

命令行生成

创建项目时，已经默认自动生成：cable.coffee、channel.rb 和 connection.rb.

使用：

```
rails generate channel NAME [method method] [options]
```

生成两文件："name_channel.rb" 和 "name.coffee"

生成的 `.rb` 文件有默认方法：subscribed 和 unsubscribed，及其它自定义的方法。
生成的 `.coffee` 文件有默认操作：链接、断开链接、接收到数据，及其它自定义操作。

依赖

自身：

```
  s.add_dependency 'actionpack', version
```

其它：

```
  s.add_dependency 'nio4r',            '~> 1.2'
  s.add_dependency 'websocket-driver', '~> 0.6.1'
```

当然了，redis 也是必须的，尽量这里没有声明。

Engine

实现方式是：Engine

其它

websockets + threads

own server process

not need to be a multi-threaded application server

Rack socket

Action View 渲染相关

包括"渲染器(n)"和"渲染(v)"，即：Renderer & Rendering.

每次都要指定模板文件，并且说一遍渲染，不麻烦吗？
因为模板文件，都在同一目录下，所以我们不必这么麻烦。引进 Action View 的子模块 Rendering 后我们设置默认的目录即可，并且能自动帮我们"说一遍渲染"。

分为 4 大块：渲染器、模板、上下文和查找模板对象。(以下只是粗略的分类)

1) 渲染器

Template Renderer
模板渲染器

Streaming Template Renderer
流模板渲染器
为了支持"streaming 流"！

Renderer
XML 渲染器

Partial Iteration
局部模板迭代

Partial Renderer
局部模板渲染器

Abstract Renderer
抽象出来的渲染器

2) 模板(Template)

对应文件及同名目录。

3) 上下文(view_context)

Rendering
渲染(动词)

~~Context~~
部分上下文
它是 `view_context` 很小的组成部分。

~~Output Flow & Streaming Flow~~
输出流

~~Output Buffer & Streaming Buffer~~
输出缓冲区

4) 查找模板对象(lookup_context)

View Paths
视图文件所在目录

~~PathSet~~
路径集(供 View Paths 使用)

DetailsKey
类似 cache key

DetailsCache
为 Details 做缓存

Lookup Context
查找上下文

Base - 为渲染打基础

不同于其它几个组件，对外提供的接口是 Action View 模块本身，而不是 ActionView::Base 类。

```
module ActionView
  class Base
    attr_accessor :view_renderer
    attr_internal :config, :assigns

    delegate :lookup_context, :to => :view_renderer
    delegate :formats, :formats=, :locale, :locale=, :view_paths
, :view_paths=,
             :to => :lookup_context

    def initialize(context = nil, assigns = {}, controller = nil
, formats = nil)
      @_config = ActiveSupport::InheritableOptions.new

      if context.is_a?(ActionView::Renderer)
        @view_renderer = context
      else
        lookup_context = context.is_a?(ActionView::LookupContext
) ?
          context : ActionView::LookupContext.new(context)
        lookup_context.formats  = formats if formats
        lookup_context.prefixes = controller._prefixes if contro
ller
        @view_renderer = ActionView::Renderer.new(lookup_context
)
      end

      assign(assigns)
      assign_controller(controller)
      _prepare_context
    end
  end
end
```

因为，类和对象的概念，对于视图来说重要性没有那么大。

`view_context` 是 ActionView::Base 的实例对象。

```
self.class.superclass
=> ActionView::Base
```

```
Class.new(ActionView::Base) do
  if routes
    include routes.url_helpers(supports_path)
    include routes.mounted_helpers
  end

  if helpers
    include helpers
  end
end
```

在 ActionView::Rendering 里。

怎么传递实例变量的？

有渲染才有传递实例变量这么一说。

```
# action_view/rendering.rb
def view_context
  view_context_class.new(view_renderer, view_assigns, self)
end
```

```
# abstract_controller/rendering.rb
def view_assigns
  # Rails 自带 & 自己管理的一些实例变量
  protected_vars = _protected_ivars
  # 此方法由 Ruby 提供，通过它我们可获取 Controller 里我们定义的一些实例
变量
  variables      = instance_variables

  # ...
end
```

初始化 View 的时候，会指派实例变量：

```
# action_view/base.rb
def assign(new_assigns) # :nodoc:
  @_assigns = new_assigns.each { |key, value| instance_variable_
set("@#{key}", value) }
end
```

非 Rails 是如何渲染的

先看例子，然后看看非 Rails 是如何渲染的，需要哪几个重要元素。

标准库 ERB 举例

```
require "erb"

# build data class
class Listings
  PRODUCT = { :name => "Chicken Fried Steak",
              :desc => "A well messages pattie, breaded and frie
d.",
              :cost => 9.95 }

  attr_reader :product, :price

  def initialize(product = "", price = "")
    @product = product
    @price = price
  end

  def build
    b = binding
    # create and run templates, filling member data variables
    ERB.new(<<-'END_PRODUCT'.gsub(/^\s+/, ""), 0, "", "@product"
).result b
      <%= PRODUCT[:name] %>
      <%= PRODUCT[:desc] %>
    END_PRODUCT
    ERB.new(<<-'END_PRICE'.gsub(/^\s+/, ""), 0, "", "@price").re
sult b
      <%= PRODUCT[:name] %> -- <%= PRODUCT[:cost] %>
      <%= PRODUCT[:desc] %>
    END_PRICE
  end
end

# setup template data
listings = Listings.new
listings.build

puts listings.product + "\n" + listings.price
```

输出结果

```
Chicken Fried Steak
A well messages pattie, breaded and fried.

Chicken Fried Steak -- 9.95
A well messages pattie, breaded and fried.
```

上面的例子，也许没能体现环境的变化，再来一个：

```
require 'erb'

# 准备条件
class User
  attr_accessor :name

  def initialize(name)
    @name = name
  end
end

# 设置实例变量
@user = User.new("Kelby")

# 打包环境
binder = self.send(:binding) # 调用 :binding 私有方法

# 模板
template = "Hello, <%= @user.name %>"

# 渲染器(名词)
render = ERB.new(template)
# 渲染(动词)
render.result(binder)

# 输出结果
=> "Helo, Kelby"
```

在这里，重要元素有：渲染器(名词)对应着 `render` ，上下文对应着 `binder` ，渲染(动词)对应着 `result` .

Rails 里，对应的有：渲染器(名词)对应着 `render` ，上下文对应着 `view_context` ，渲染(动词)对应着 `render` .
Rails 里模板文件众多，而且可以嵌套使用，为了方便查找模板内容，引入了 `lookup_context` 的概念。

Rails 使用的是 Erubis

Rails 用的是 gem 'erubis'，不是 Ruby 标准库里的 ERB，不过它们原理类似，不再深入。

链接

Binding
ERB
Erubis

Rails 是如何渲染的

别忘记了在 YourController#your_action 里，除了有 `render` 方法外，还有 N 多和渲染相关的方法！有它们就完全可以做渲染任务了。

在某个 action 里打断点，然后：

```
self.respond_to? :render
# => true

self.respond_to? :lookup_context
# => true

self.respond_to? :view_renderer
# => true

self.respond_to? :view_context
# => true

self.respond_to? :view_assigns
# => true
```

> Note: 在 YourMailer#your_action 里打断点，结果一样。

我们调用的是 `render` 方法时，它会调用到 `lookup_context` 和 `view_renderer`

而 `lookup_context` 可以帮我们找到要渲染的模板。

渲染器 + 模板内容 + 上下文

```
view_renderer.new(@lookup_context).render(context, options, bloc
k)
```

`view_renderer` 指的是渲染器，也就是：TemplateRenderer、PartialRenderer 或 StreamingTemplateRenderer.

`@lookup_context` 用来查找模板的。

`context` 指的是上下文，对应上面的 `view_context` .

通过 `self.view_assigns` 或 `self.view_context.assigns` 可以查看，我们在 Controller 传递给 View 的实例变量(它们只是上下文内容里的一部分)。

Rendering

承上启下的作用：

```
            ActionController::Rendering
                      |
                      V
            AbstractController::Rendering
                      |
                      V
            ActionView::Rendering
                      |
                      V
            ActionView::Renderer
```

Action Controller 和 Abstract Controller 可以调用这里渲染相关的方法。

实例方法：

```
view_context
view_renderer

render_to_body

rendered_format

view_context_class
```

`view_renderer` 是 ActionView::Renderer 的实例对象。

`view_context` 是 ActionView::Base 的实例对象。

`render_to_body` 上面几个方法中，惟一的动词，会执行渲染程序。

view_context 使用举例：

通过 `view_context` 可以在 Controller 里调用 Helper 里的方法。

举例一：

```
module ApplicationHelper
  def fancy_helper(str)
    str.titleize
  end
end

class MyController < ApplicationController
   def index
     @title = view_context.fancy_helper "dogs are awesome"
   end
end
```

举例二：

```
# app/controllers/posts_controller.rb
class PostsController < ActionController::Base
  def show
    # ...
    tags = view_context.generate_tags(@post)
    email_link = view_context.mail_to(@user.email)
    # ...
  end
end

# app/helpers/posts_helper.rb
module PostsHelper
  def generate_tags(post)
    "Generate Tags"
  end
end
```

类方法：

```
view_context_class
```

并且，ActionController 和 AbstactController 有同名方法，根据 Ruby 的继承规则，它们也可以调用这里的方法。

当然，还有其它方法，但不在此列举。

渲染器介绍

渲染器需要有上下文和模板对象，才能顺利的工作。

(但它不直接和模板打交道？谁和模板打交道？)

最后

根据要渲染的内容，render 还有不少的可选参数，比如：:partial、:template、:inline、:file 和 :text，使用的时候需要根据情况挑选使用。

Renderer - 渲染的入口

渲染的入口。(它只是调用，真正的渲染工作不是它做的)

`render` 或 `render_body` 方法加上它们的参数，决定了使用 Template Renderer、Partial Renderer 还是 Streaming Template Renderer.

```
# Main render entry point shared by AV and AC.
render(context, options)

# Render but returns a valid Rack body.
render_body(context, options)

# 下面这两个方法，没有对外提供的接口，不要直接使用它们！

# Direct accessor to template rendering.
render_template(context, options)

# Direct access to partial rendering.
render_partial(context, options, &block)
```

> Note: 说明一下，Streaming Template Renderer 用得很少，本说明后文不再重复。

基类 Abstract Renderer

具体某个渲染器的抽象类、基类，定义了接口，并提供了几个保护方法给子类使用。

目前，只有 3 个渲染器，Template Renderer、Partial Renderer 和 Streaming Template Renderer.

接口：

```
render
```

保护方法：

```
extract_details
prepend_formats
```

除上述外，还有：

```
delegate :find_template, :template_exists?, :with_fallbacks, :wi
th_layout_format,
         :formats, :to => :@lookup_context
```

子类 **Template Renderer**

普通的模板渲染器，直接继承于 Abstract Renderer.

用到了 template 里的东西。

要考虑 layout.

借助了 @lookup_context.

子类 **Partial Renderer & Collection Caching**

局部模板渲染器，直接继承于 Abstract Renderer.

可以渲染一个集合，此时借助了 Partial Iteration.

借助了 @lookup_context.

子类 Streaming Template Renderer

流模板渲染器，直接继承于 Template Renderer，间接继承于 Abstract Renderer.

重写了父类的 render_template 方法。

Streaming Template Renderer 用得很少，不做过多介绍。

Template 内容

模板对象的内容，及其所携带的信息。

```
ERBHandler = ActionView::Template::Handlers::ERB.new

body = "<%= hello %>"
details = { format: :html }

def new_template(body = "<%= hello %>", details = { format: :html
 })
  ActionView::Template.new(body, "hello template",
                           details.fetch(:handler) { ERBHandler
},
                           {:virtual_path => "hello"}.merge!(det
ails))
end

@template = new_template
@template = new_template("<%= apostrophe %>")
@template = new_template("<%= apostrophe %> <%== apostrophe %>",
format: :text)
@template = new_template("<%= hello %>",
                         :handler => ActionView::Template::Handl
ers::Raw.new)
```

当然，这些我们平时都接触不到，知道有这么回事即可。

```
attr_accessor :locals, :formats, :variants, :virtual_path

attr_reader :source, :identifier, :handler, :original_encoding,
:updated_at

# 看看新建模板，要求是什么
def initialize(source, identifier, handler, details)
  if handler.respond_to?(:default_format))
    format = details[:format] || (handler.default_format
  end

  @source            = source
  @identifier        = identifier
  @handler           = handler
  @compiled          = false
  @original_encoding = nil
  @locals            = details[:locals] || []
  @virtual_path      = details[:virtual_path]
  @updated_at        = details[:updated_at] || Time.now
  @formats           = Array(format).map { |f| f.respond_to?(:ref
) ? f.ref : f  }
  @variants          = [details[:variant]]
  @compile_mutex     = Mutex.new
end
```

template 本文件下的内容

```
encode!

inspect

refresh

render

supports_streaming?

type
```

这里的 `render` 会调用方法，进而渲染对应的模板。

```
attr_accessor :locals, :formats, :variants, :virtual_path

attr_reader :source, :identifier, :handler, :original_encoding,
:updated_at
```

template 目录

包含了：error、handlers、html、resolver、text、types 等。
其中 handlers 又包含了：builder、erb、raw 等。

其中，Html 和 Text 这两个类为模板(较简单)，handlers 下面的 Builder、ERB 和 Raw 3 个类也是模板(较复杂)。

从上述可知，Renderer 其实也没做什么，处理工作交给 Template 进行处理。

```
Renderer(最外层的接口)
       |
       V
各个 Renderer(中间处理)
       |
       V
Template(最底层的处理)
```

template/ 目录里的各个 handler，执行的是与 Rails 环境无关的解析、渲染工作。

Template
视图文件

Type
类型(也就是 format)

Text & HTML
(什么也不是)

Resolver
解析器

Path Resolver
路径解析器

File System Resolver
文件系统解析器

Optimized File System Resolver
优化的文件系统解析器

Fallback File System Resolver
回溯的文件系统解析器

Error
各种错误

Handlers
处理器，包括：

- Raw
  原生的处理器
- Erubis
  Erubis 处理器
- ERB
  ERB 处理器
- Builder
  XML 处理器

Lookup Context

包含了一些基本信息，用于查找模板。

由原来的 Template::Lookup 更改而来，其实名字叫 Lookup Template Context 反正更合适。

lookup_context 内容

基本信息，包括以下 7 项内容：

```
def initialize(view_paths, details = {}, prefixes = [])
  @details, @details_key = {}, nil
  @skip_default_locale = false
  @cache = true
  @prefixes = prefixes
  @rendered_format = nil

  self.view_paths = view_paths
  initialize_details(details)
end

# 也就是：

@cache=true,
@details= { ... },
@details_key=nil,
@prefixes=[],
@rendered_format=nil,
@skip_default_locale=false,
@view_paths= #<ActionView::PathSet:0x007ff7250a0fe0 ...
```

在 ActionView::Base 里有：

```
delegate :formats, :formats=, :locale, :locale=, :view_paths, :v
iew_paths=,
         :to => :lookup_context
```

lookup_context 举例

查看应用根目录下文件及目录：

```
✗ ls
Gemfile       README.rdoc  app  blorgh  config.ru  lib  public
tmp
Gemfile.lock  Rakefile     bin  config  db         log  test
vendor
```

创建一个简单的 lookup_context

```
FIXTURE_LOAD_PATH = File.join(File.dirname(__FILE__), '../fixtur
es')
# => "./fixtures"

require 'action_view'
# => true

# 仍然只包含 7 项内容
lookup_context = ActionView::LookupContext.new(FIXTURE_LOAD_PATH
, {})
=> #<ActionView::LookupContext:0x007ff725058088
 @cache=true,
 @details= { ... },
 @details_key=nil,
 @prefixes=[],
 @rendered_format=nil,
 @skip_default_locale=false,
 @view_paths= #<ActionView::PathSet:0x007ff7250a0fe0 ...
```

真实情况远比这复杂， `@view_paths` 下有很多的内容。

实例方法

```
attr_accessor :prefixes, :rendered_format
mattr_accessor :fallbacks
mattr_accessor :registered_details

formats=

skip_default_locale!

locale
locale=

with_layout_format
```

类方法

```
register_detail
```

放到变量 registered_details 里，并生成 default_x 等方法。

```
# 默认已经注册有：locale、formats、variants 和 handlers.

register_detail(:locale) { ... }
register_detail(:formats) { ... }
register_detail(:variants) { ... }
register_detail(:handlers) { ... }
```

View Paths

实例方法：

```
find & find_template
exists? & template_exists?

find_all

view_paths=

with_fallbacks
```

`find` 很重要的方法，用来查找模板(Template)。

`template_exists?` 使用举例，在视图里检查局部模板是否存在：

```
template_exists?("form", "posts", true)
```

> Note: 区别于【ActionView::ViewPaths】对应的模块。

~~Details Key~~

```
get

clear
```

~~Details Cache~~

```
details_key

disable_cache
```

view_context

渲染相关里的上下文 **- ActionView::Base** 的实例对象。

`view_context` 就是 ActionView::Base 的实例对象。
`_view_context_class` 默认就是 ActionView::Base

view_context 内容

```
BlogsController.new.view_context
=> #<#<Class:0x007ff084a6ba90>:0x007ff07b745518
 @_assigns={"_routes"=>nil},
 @_config={},
 @_controller= BlogsController:0x007ff084a6bba8
 @_request=nil,
 @_routes=nil,
 @output_buffer=nil,
 @view_flow=#<ActionView::OutputFlow:0x007ff07b757010 @content={
}>,
 @view_renderer= ActionView::Renderer:0x007ff07b745cc0
 @virtual_path=nil>
```

1) view_context.controller

```
@_controller=
#<BlogsController:0x007ff084a6bba8
 @_action_has_layout=true,
 @_config={},
 @_headers={"Content-Type"=>"text/html"},
 @_lookup_context= ActionView::LookupContext:0x007ff07b7474a8
 @_prefixes=["blogs", "application"],
 @_request=nil,
 @_response=nil,
 @_routes=nil,
 @_status=200,
 @_view_context_class=#<Class:0x007ff084a6ba90>,
 @_view_renderer= ActionView::Renderer:0x007ff07b745cc0
```

2) view_context.view_renderer

```
@view_renderer=
 #<ActionView::Renderer:0x007ff07b745cc0
  @lookup_context= ActionView::LookupContext:0x007ff07b7474a8
```

类似概念 controller context

在 Controller 里，除了实例变量，我们还可以有其它方法传递内容给 View，两者方式类似。

```
class BasicController < ActionController::Base
  # 1) 只引入对应模块
  include ActionView::Context

  # 2) 调用对应模块里的方法
  before_filter :_prepare_context

  def hello_world
    @value = "Hello"
  end

  protected
  # 3) 更改 view_context
  # 默认它是 ActionView::Base 的实例对象
  def view_context # STEP 3
    self
  end

  # 在 View 里可以调用此方法
  def __controller_method__
    "controller context!"
  end
end
```

```
# app/views/basic/hello_world.html.erb
<%= @value %> from <%= __controller_method__ %>
```

因为 request/response 不是完整的，不能在浏览器里通过 url 访问，使用以下命令验证：

```
curl http://dev.shixian.com:3000/hello_world
```

返回：

```
# app/views/basic/hello_world.html.erb
Hello from controller context!
```

> Note: 本示例仅为了有助于理解 view_context 及理解 Rails 是如何渲染内容的，不可用于实际项目。

参考

Design Principles behind

~~Context~~

部分上下文内容。

从 Base 里抽取出来的，提供了 @output_buffer, @view_flow 和 @virtual_path

为了 Action Controller 可以创建 ActionView::Base 的实例对象，然后进行渲染模板的工作。

不要被名字欺骗了，它只是 `view_context` 很小的组成部分。

~~Output Flow & Streaming Flow~~

输出流

一般的渲染器用的是"Output Flow"；
当使用的渲染器是"Streaming Template Renderer"的时候，"Streaming Flow"就能派上用场了。

View Paths

把路径信息放到 PathSet，之后的 lookup_context 需要它们。

类方法：

```
append_view_path
prepend_view_path

view_paths
view_paths=
```

`append_view_path(path)` 追加路径到当前 Controller 所在的 view paths 里。

当我们的自己创建或者使用的 gem 引入的文件目录结构和 Rails 默认不一样，但且希望渲染到视图里的模板(局部模板)，就会报错，此方法可以帮助我们。

`prepend_view_path` 和 append_view_path 方法类似。

实例方法：

```
append_view_path
prepend_view_path

details_for_lookup

lookup_context
```

`lookup_context` 之前已经介绍过，很重要的一个方法。

其它：

```
delegate :template_exists?, :view_paths, :formats, :formats=, :l
ocale, :locale=,
         :to => :lookup_context
```

这里的内容，原来是放在 abstract_controller/rendering.rb 文件里的，后来单独成 view_paths.rb，再后来从 Abstract Controller 移到 Action View.

> Note: 区别于【ActionView::LookupContext::ViewPaths】对应的模块。

其它

TODO

Output Buffer & Streaming Buffer

普通字符串使用 `html_safe` 后就变成了 Safe Buffer 的实例对象。

Output Buffer 继承于 Safe Buffer，它封装了一些方法，Rails 视图里使用的就是它。

`html_safe?` 和 Safe Buffer 并不是一对一关系，所以会出现 Safe Buffer 实例对象的 html_safe? 为 false 的情况(这同样影响了子类 Output Buffer)；并且视图里字符串拼接(concat)对性能也会有影响。

基于种种原因，和 Output Buffer 相对的引入了 Streaming Buffer，但它以'流'的形式传递数据到客户端，并且它不继承于 Safe Buffer，所以绕开了一些问题。

举例：
使用"Capture Helper"，会用到"Output Buffer"；
而使用"Streaming Template Renderer"，则会用到"Streaming Buffer"。

参考

YAGNI methods slow us down
ActionView Safe Buffer

Action View 提供的辅助方法

下面 helper 分类，只是为了方便理清它们的结构。实际过程中，可交叉使用，能达到目的即可，并且直接写 HTML 也是允许的。

有的方法根据其参数，可归于多个分类。如：表单元素和通用元素(非表单元素)。

因为 Rails 背后会把所有 helper 方法(函数)都会被放进同一个 module 里，所以它们之间互相调用。

与表单直接相关的辅助方法

按照代码结构上，可分为四类：

- Form Builder
- Form Helper
- Form Options Helper
- Form Tag Helper

按照使用方式不同，又可分为三类：

- Form Builder 对 Model 依赖最重，表单对象几乎等价于 record 对象。
- Form Helper 和 Form Options Helper 次之。
- Form Tag Helper 对 Model 依赖最轻，没有表单对象的概念，操作上几乎等价于 HTML（其实就是语法糖）。

Form Builder 和 Form Helper 在方法、函数上基本是对应的。

Form Builder 是面向对象，Form Helper 是面向函数。

Form Builder

Form Builder 是面向对象。

你要修改的是 record 对象，途径是通过表单实现。 所以，record 对象和表单就必需有某种联系。 在这里，这种联系由 Form Builder 及其实例对象完成。

源代码里，有 **3** 个来源：

```
date_helper.rb
form_helper.rb
form_options_helper.rb
```

大部分是封装 @template

大部分以 f.xxx 的形式调用。

> Note: f 就是 FormBuilder 的实例对象，所以可以调用 FormBuilder 的实例方法。

构建器怎么来？

```
# form_for 和 fields_for 调用 instantiate_builder

# 再调用
default_form_builder
  builder = ActionView::Base.default_form_builder
  ... ...
end

ActiveSupport.on_load(:action_view) do
  cattr_accessor(:default_form_builder) { ::ActionView::Helpers:
:FormBuilder }
end

# 简单理解 builder = FormBuilder

# 最后调用
builder.new(object_name, object, self, options)
```

专用于 **f.xxx** 调用。

```
button

check_box

collection_check_boxes
collection_radio_buttons
collection_select

date_select
datetime_select

fields_for

file_field

grouped_collection_select

hidden_field

label

radio_button

select

submit

time_select
time_zone_select
```

扩展 Form Builder

使用:

```
<%= form_for @person do |f| %>
  Name: <%= f.text_field :name %>
  Admin: <%= f.check_box :admin %>
<% end %>
```

这里的 `f` 是 Form Builder 的实例对象，所以可以直接调用 FormBuilder 提供的方法。

扩展:

你可以继承于 Form Builder，然后构建和表单相关的 helper 方法，举例：

```
class MyFormBuilder < ActionView::Helpers::FormBuilder
  def div_radio_button(method, tag_value, options = {})
    @template.content_tag(:div,
      @template.radio_button(
        @object_name, method, tag_value, objectify_options(optio
ns)
      )
    )
  end
end
```

```
<%= form_for @person, :builder => MyFormBuilder do |f| %>
  I am a child: <%= f.div_radio_button(:admin, "child") %>
  I am an adult: <%= f.div_radio_button(:admin, "adult") %>
<% end -%>
```

Form Helper

Form Helper 是面向函数。

相对与 Form Builder 它更灵活，因为它没有限定对象。有时候我们'需要'这样做，而有时候我们没'必要'。

大部分封装 Tags::Xxx

名字 Form Helper 起得有一点不合适。

部分方法需要的参数 "object_name"，并不特指 record 对象，也可以是非 record 对象。

```
check_box
color_field

date_field
datetime_field
datetime_local_field

email_field

fields_for

form_for

label

file_field

hidden_field

month_field

number_field

password_field
phone_field

radio_button
range_field

search_field

telephone_field
text_area
text_field
time_field

url_field

week_field
```

form_for 和 fields_for 是另类，它们都可用于构建表单：

`form_for` 和 record 对象关联比较大。
`fields_for` 区别于 form_for，与 record 对象直接关联不大。

Form Options Helper

多用于处理集合。

常见于两个对象之间。

分为目的 和 手段。

比如：select_tag 是目的，options_for_select 是手段。

名字 Form Options Helper 起得有一点不合适。

部分方法需要传递参数 "object"，并不特指 record 对象，可以是非 record 对象。

最常用的几个方法

关键词：select

```
select

collection_select

# 子关键词：optgroup
grouped_collection_select

# 和 time_zone 有关
time_zone_select
```

关键词：option

```
options_for_select

options_from_collection_for_select
```

相对而言，几个不太常用的方法

关键词：option

```
# 和 time_zone 有关
time_zone_options_for_select
```

关键词：optgroup

```
grouped_options_for_select

option_groups_from_collection_for_select
```

关键词：checkbox

```
collection_check_boxes
```

关键词：radio

```
collection_radio_buttons
```

Form Tag Helper

最接近 HTML 代码。

没有对象(不用对应 model)及相关概念。

封装 tag 或 content_tag 而来。

名字 Form Tag Helper 起得非常不合适。
这里的方法与表单及 record 对象都没有直接相关，它只用于生成 HTML 代码。也就是说它们并不依赖于表单元素和 record 对象。把它们放到此章节，完全是因为它们所在的模块名字带 "Form".

button 标签

```
button_tag
```

input 标签

```
check_box_tag

color_field_tag

date_field_tag
time_field_tag
datetime_field_tag
datetime_local_field_tag

email_field_tag

file_field_tag

hidden_field_tag

image_submit_tag

month_field_tag
```

```
number_field_tag

password_field_tag

phone_field_tag & telephone_field_tag

radio_button_tag

range_field_tag

search_field_tag

submit_tag

text_field_tag

url_field_tag

utf8_enforcer_tag

week_field_tag
```

fieldset 标签

```
field_set_tag
```

label 标签

```
label_tag
```

select 标签

```
select_tag
```

textarea 标签

```
text_area_tag
```

form 标签

```
form_tag
```

上面并不是表单可以使用的所有方法，有一些是动态定义的。

```
default_form_builder = ::ActionView::Helpers::FormBuilder
builder = default_form_builder

object_name = 'product'
object = nil
options = {}

f = builder.new(object_name, object, self, options)
=> #<ActionView::Helpers::FormBuilder:0x007feaa896bc80
 @default_options={},
 @index=nil,
 @multipart=nil,
 @nested_child_index={},
 @object=nil,
 @object_name="product",
 @options={},
 @template=main>
```

与表单非直接相关的辅助方法

Rails 提供了很多的 helper 供我们使用，处理 asset、date、form、number 和 record 对象等。默认它们可以在所有模板上使用。

基于 Tag 实现的 helper, 部分可以到 action_view/helpers/tags 目录下找到相应文件，里面的 render 方法里面有常用的可选参数。

对设置默认值等，特别有帮助。不用写 html，也不用额外添加变量。

当和 record 对象没有关联时，尽量使用 x_tag 的形式。

区别于表单元素，这里的 helper 不用束缚于表单，比较通用。

另，表单元素其实也可以脱离表单来使用。所以，上面有提到 FormXxx 模块名起得不太合适(我又啰嗦了)。

Tag Helper

几乎所有生成 HTML 元素的 helper 方法，都封装它们而来。

```
content_tag

tag
```

`content_tag` 用于生成一个闭合的 HTML 标签。很多 helper 方法，都是封装于它。
`tag` 用于生成一个打开的 HTML 标签。很多 helper 方法，都是封装于它。

```
cdata_section

escape_once
```

Url Helper

可直接转化成 HTML：

```
link_to
button_to
mail_to
```

涉及逻辑判断的方法：

```
link_to_if
link_to_unless
link_to_unless_current

current_page?
```

Asset Tag Helper

得到的是包含 asset 在内的 HTML 代码。

```
javascript_include_tag
stylesheet_link_tag

image_tag
image_alt

favicon_link_tag

auto_discovery_link_tag

audio_tag
video_tag
```

Cache Helper

缓存。

```
cache
```

```
cache_if
cache_unless

cache_fragment_name
```

片段缓存和 Cache Key

1) 页面上的静态内容

2) 页面上的动态内容

3) 页面上的内容，在 model 层面有关联

4) 页面上的内容有嵌套关系

使用片段缓存几点原则

1. 缓存由动态内容和静态内容两部分构成。
2. 动态内容的 cache_key 由我们指定；
    i. 没有嵌套的情况下，如果动态内容不指定 cache_key，则自己的动态内容永远不会更新(例外见最后)；
    ii. 有嵌套的情况下，如果动态内容不指定 cache_key，则自己的动态内容 & 孩子的动态内容永远不会更新(例外见最后)；
3. 没有嵌套的情况下，有且只有自己的 cache_key 更新，动态内容才更新；
4. 有嵌套的情况下，有且只有自己的 cache_key 更新 & 父亲的 cache_key 更新，动态内容才更新；
5. 动态内容的更新，不影响静态内容的部分；
6. (各动态内容的 cache_key 是独立的，自己及其父亲、兄弟、孩子的 cache_key 没有依赖关系)
7. 无论哪的静态内容更改，有且只有重启后更新，不存在(也不用考虑)嵌套的问题；
8. 静态内容的更新，不影响动态内容的部分(例外见最后)；
9. 例外：动态内容没有指定 cache_key，只有静态内容同时更新，并且重启，动态内容才会更新。

说一说 Cache Key

1) record 的 cache_key

2) helper 方法 cache(name = {}, options = nil, &block) 这里的 name

3) 内存数据库 key

不严格区分的话，它们都可以叫做"Cache Key"

但，你可以把它们区分开来：

```
post.cache_key
# => "posts/1-20140921032815201680000”
```

```
<% cache [ 'a_post', post ] do %>
  ... ...
<% end %>
```

```
views/a_post/posts/1-20140921032815201680000/9746fd05c8428f79996
81aa804071e9a
(路径    helper方法cache的name部分    静态内容md5)
```

特点：

1 由 record 的"id + updated_at时间戳"组成，所以 record 更新，cache_key 会更新(update_column等情况不讨论)

2 由我们指定，所以理论上可以随便写。没有嵌套缓存的情况下，一般可以直接使用 1；有嵌套的情况下可以用 touch 更新父亲资源或者使用一个组合，如: cache [current_user, post]

3 根据 2 生成 (但不等于 2)，要求唯一 (要不然就没意义了)

2 每次请求都要计算

2 计算之后和 3 对比是否匹配。不匹配则生成新的内容，渲染页面；匹配则返回，渲染页面

2 不匹配，那么意味着之前的内容过期了，过期的内容还是继续存在的数据库里的。(这里的过期不等于被删除)

2 过期的内容可以由我们手动删除，如：Rails.cache.clear；或让数据库自动删除，如 `config.cache_store = :redis_store, 'redis://localhost:6379/5/cache', { expires_in: 90.minutes }` 这里数据最多只能在 redis 里保存 redis 每隔 90 分钟，到期的缓存数据会被删除。(这里的到期等于被删除)

几点建议

缓存主要有两种方式过期。

1. 调用 cache(name = {}, options = nil, &block) 的时候把会引起数据变化的元素都放到 name 里
2. 后续更新一个对象的时候，touch 其关联对象

- 嵌套太深不适合使用缓存(不超过 4 层)
- 涉及太多不同类型的 record 对象时，不推荐使用缓存
- 弱关联(非 has_many, has_one, belongs_to)对象，不适合放在一起做缓存

嵌套太深或 record 对象太多, 或几个对象之间属于弱关联，无论使用哪种缓存过期机制对于维护都是恶梦。

Controller Helper

委托 **Controller** 方法给 **View** 使用。

```
delegate :params, :session, :cookies, :flash,
         :response, :headers, :action_name, :controller_name, :c
ontroller_path,
         :request_forgery_protection_token,
         :to => :controller
```

效果和使用 `helper_method` 类似。

ActionView::Base initialize 时就会引入。(其实，它就是从 Base 里抽取出来的)

Asset Url Helper

仅得到 asset 所在的路径。

```
image_path & path_to_image
image_url  & url_to_image

javascript_path & path_to_javascript
javascript_url  & url_to_javascript

stylesheet_path & path_to_stylesheet
stylesheet_url  & url_to_stylesheet

video_path & path_to_video
video_url  & url_to_video

asset_path & path_to_asset
asset_url  & url_to_asset

audio_path & path_to_audio
audio_url  & url_to_audio

font_path & path_to_font
font_url  & url_to_font

compute_asset_extname
compute_asset_host
compute_asset_path
```

Csrf Helper

```
csrf_meta_tag & csrf_meta_tags
```

Capture Helper

存储代码块，`yield` 调用。

```
content_for
content_for?

capture

provide
```

其中 `capture` 和 `content_for` 作用类似，只是定义和调用稍有不同。

使用举例：

```
# 定义
<% content_for :navigation do %>
  <li><%= link_to 'Home', action: 'index' %></li>
<% end %>

# 调用
<ul><%= yield :navigation %></ul>

# 或
<ul><%= content_for :navigation %></ul>
```

Debug Helper

```
debug
```

使用 gem 'pry-rails' 后，很少有使用这个方法。可以在要打断点的地方调用 `binding.pry`

或者，使用 Rails 默认使用的 `debugger` 打断点。

Date Helper

```
date_select
datetime_select
time_select

distance_of_time_in_words
time_ago_in_words & distance_of_time_in_words_to_now

select_date
select_time
select_datetime

select_year
select_month
select_day
select_hour
select_minute
select_second

time_tag
```

抽取出了 DateTime Selector，用来处理这里和 select_x 相关的方法。

JavaScript Helper

```
j & escape_javascript

javascript_tag
```

Number Helper

```
number_to_currency
number_to_human
number_to_human_size
number_to_percentage
number_to_phone

number_with_delimiter
number_with_precision
```

Output Safety Helper

```
raw

safe_join
```

Rendering Helper

渲染：调用渲染器进行渲染工作(这里全部是调用，并且主要是渲染局部模板)。

```
render
```

会用一个章节详细描述其相关内容。

~~Record Tag Helper~~

```
content_tag_for

div_for
```

增加复杂度，还不如直接使用 HTML 并嵌套 ERB 代码。

已被废除。

Sanitize Helper

```
sanitize
sanitize_css

strip_links
strip_tags
```

这里的"消毒"，使用的是 Rails Html Sanitizers 这里只是封装，并提供接口。

Translation Helper

```
t & translate
l & localize
```

翻译和本地化。

Text Helper

```
truncate      # 截取某个长度的文本

concat        # 求值

cycle         # 循环
current_cycle # 循环内当前值
reset_cycle   # 中断循环，从头开始

excerpt       # 引用、摘录(只显示某个词及其附近的语句，其余用 ... 代替)
highlight     # 高亮(默认是加 mark )
pluralize     # 复数形式

safe_concat   # 在 Active Support 里也提供了 safe_concat 方法，如果可用则用；否则用 concat
simple_format # 文本简单格式化
word_wrap     # 限制长度
```

Atom Feed Helper

不同于其它信息，Atom 信息不能用 ERB 或其它模板语言创建。但通过这个方法，却能构建 Atom 订阅信息。

```
atom_feed
```

抽取出了 Atom Feed Builder 和 Atom Builder，用来处理这方面的工作。

~~Active Model Instance Tag~~

```
content_tag

error_message

error_wrapping

object

tag
```

其它

在 **model** 里，使用 **helper** 方法？

1) 直接使用：

```
OrdersController.helpers.order_number(@order)
ApplicationController.helpers.helper_method
ActionController::Base.helpers.sanitize(str)
```

2) 先引入，再使用：

```
include ActionView::Helpers::DateHelper
include ActionView::Helpers
```

方法太多，看不过来？

有的 Rails 内置方法，难懂、难用，所以可以不用。以 Action View 举例说明：

1. 我们需要看 API，才知道怎么传递参数，有什么参数；
2. 直接写 HTML，大脑不必转一圈；
3. 前端工程师不懂 Ruby，就看不懂；
4. 维护的时候也遇到上述 3 点困境。

Action View 其它类和模块

下面这些方法，和渲染没有直接关联，并且严格意义上讲不属于 helper 方法。

Record Identifier

根据所传递的对象，生成能代表其身份的"字符串"。

```
dom_class
dom_id
```

可配合其它 helper 一起使用，使用举例：

```
dom_class(post)   # => "post"
dom_class(Person) # => "person"

# 带前缀
dom_class(post, :edit)   # => "edit_post"
dom_class(Person, :edit) # => "edit_person"
```

```
dom_id(Post.find(45))       # => "post_45"
dom_id(Post.new)            # => "new_post"

# 带前缀
dom_id(Post.find(45), :edit) # => "edit_post_45"
dom_id(Post.new, :custom)    # => "custom_post"
```

实现它们时直接使用了"字符串求值的"方式，并且用到了 ActiveModel::Model 里的方法。

> Note: 它既没有对应也没有生成 HTML 标签。

Routing Url For

```
url_for
```

它生成的内容是 url，区别于 link_to 等生成的是链接。

可选参数的类型可以是：Hash、String、nil、:back、Array 等，根据不同参数，可能会用到 ActionDispatch::HelperMethodBuilder 里的东西。

> Note: 有多个 url_for 方法，这个是 View 里用到的。

Layouts

`layout`

布局，影响渲染的效果，但和渲染没有直接关联。

```
# String
class InformationController < BankController
  layout "information"
end
```

```
# Symbol
class VaultController < BankController
  layout :access_level_layout
end
```

```
# nil
class CommentsController < ApplicationController
  # 始终使用默认的 layout (首先是 comments，然后是 application 的)
  layout nil
end
```

```
# false
class TillController < BankController
  layout false
end
```

此外，还有可选参数 `:only` 和 `:except`

> Note: 没有 layout true 这种写法，会报错的。

layout 有继承关系：

```
class ApplicationController < ActionController::Base
  layout "application"
end

class PostsController < ApplicationController
  # 继承使用 "application" layout
end
```

~~Model Naming~~

Action View 里有 Model Naming 模块。

Controller 和 View 经常要处理 record 对象，比如得到单数、复数格式，得到 cache_key、route_key、param_key 等，这都需要用到 Model 里的方法。此时，可以把 record 对象转化得到 model_name，然后再进行后续操作。

ActionController::ModelNaming 和 ActionView::ModelNaming 一样，都提供了：

```
convert_to_model

model_name_from_record_or_class
```

方便我们把对象转换成 Model 或 model_name 进行处理，但一般我们不会直接使用。

~~Digestor~~

对视图进行加密，是片段缓存的组成部分。

类方法：

```
digest

tree
```

目前，Rails 里有两个地方调用了它。

1) Cache Helper 会调用。(也就是说调用 cache helper 方法时，会调用到它)

2) Action Controller 里的 Etag With Template Digest 也会调用。(可配置将模板 digest 的结果放到 etaggers 里)

实例方法：

```
digest
dependencies

nested_dependencies
```

`dependencies` 会用到 Dependency Tracker.

> Note: 有类方法 digest 和同名实例方法 digest. 一般我们直接调用的是类方法，也就是 ActionView::Digestor.digest，之后它内部处理，调用实例方法。

~~Dependency Tracker~~

供 Digestor 使用，digest 的时候可以以数组的形式传 dependencies 作为其可选参数。

提供类方法：

```
register_tracker
remove_tracker

find_dependencies
```

以数组的形式返回其依赖的模板（模板、局部模板、指定 layout 的模板）的名字：

```
def dependencies
  render_dependencies + explicit_dependencies
end
```

一个 action 模板只有一个，所以主要针对的是里面所包含的局部模板。

其它

为什么有的 **helper** 我不推荐？

以下是我的一些经验之谈，仅供参考。

- 基于其它 helper，并且功能上十分类似；
- 使用它，写出来的代码反而很丑陋；
- 和样式关联密切，或者CSS、JS也能做的；
- 不实用、或者说不通用

不要迷信 Rails 提供的 helper，有的用起来难读、难懂，还不如自己写。别忘了，设计软件的重点：好读、易维护、以全局观思考。

Active Model

我们知道 MVC 结构里，Model(模型)层与数据库关系最紧密。现在我们是把 Model(模型层)再拆分一下，把对数据库真正有操作的这部分拆分成 Active Record，把没有对数据库操作的这部分拆分成 Active Model.

做了这样的拆分之后，如果我们觉得 Active Record 不合口味(例如：想使用 NoSQL)，但又不想完全抛弃它。解决方案来了，使用 Action Model，Active Model 可以以接口的形式提供一系列方法给 model 使用。

Active Record 对于需要持久化的数据进行校验或处理，是很强大的。但如果我们不需要持久化数据，或者我们只需要很少的功能，如：提交表单数据在 Web 开发中是比较常见(联系我们，给我们发送反馈意见) - 用户提交信息，无论校验是有效还是无效，都应该得到反馈。Active Record 太重了，使用 Active Model 可以帮助我们减少复杂度。

> Note: 说明一下，Active Model 脱胎于 Active Record，它可以单独使用； 反过来，Active Record 依赖于 Active Model，不可以单独使用。

Model - 核心

这里指的是 ActiveModel::Model，它及它下面所包含的 Validations、Conversion 和 Naming、Translation 等模块是整个 Active Model 里的核心部分。

Model 做的事情包括但不限于: 校验 和 抽象 **ORM** 接口。

Conversion & Validations 是 `include` 如果没有指明 ClassMethods，则引入的是实例方法；
Naming & Translation 是 `extend` 默认引入的就已经是类方法。 注意这点小差别，对于使用上会有帮助。

Validations

在数据存入数据库存之前，有多个层面可以对数据做校验。包括客户端校验、Controller 级别的校验、Model 级别的校验和数据库本身做约束。它们各有利弊，在此不做讨论。

这里要讲的是 Model 级别的校验。

类方法

常用：

```
validate                  # 调用方式一(不需要属性，也不需要校验器；直接做校验)
validates 和 validates! # 调用方式二(需要属性，也需要校验器；间接做校验)
validates_each            # 调用方式三(需要属性，不需要校验器；直接做校验)

validates_with            # 调用方式四(不需要属性，需要校验器；作用于所有对象，间接做校验)
```

除上述方法外，还有：

```
validators # 查看所有可用的校验器
validators_on(*attributes) # 查看在某属性上使用了哪些校验器

clear_validators! # 清除校验器
```

以及：

```
attribute_method?(attribute)
```

`attribute_method?` 放在这里有点不太合适，它和校验没有多大关系。和 Attribute Methods 反而关系比较大。

经验：牢记 `validate` 发生在 `save` 之前，如果你喜欢用 `before_save` 之类的进行检验，记得加上 `return false` 或者改变习惯，用 `validate :validate_method` 类似写法进行校验。

9 个 Helper Methods

9 个校验方法，语法糖。 调用方式五(不需要属性，需要校验器；作用于某个具体对象，间接做校验)

```
validates_format_of
validates_length_of & validates_size_of
validates_absence_of
validates_presence_of
validates_exclusion_of
validates_inclusion_of
validates_acceptance_of
validates_confirmation_of
validates_numericality_of
```

由于它们封装 `validates_with` 而来，既可当做类方法进行调用。
又由于它们具体实现时继承于 EachValidator，又可以当做 `validates` 的参数使用。

`validates_confirmation_of` 会为要校验的属性生成对应的读、写方法(x_confirmation、 x_confirmation=)

如：校验 password，会生成 password_confirmation 读方法和 password_confirmation= 写方法。

实例方法

```
errors

invalid?
valid? & validate

validates_with
```

`valid?` 校验是否通过，只有一个判断规则：当前 record 对象的 errors 值是否为空 errors.empty?
所以自己写校验方法或者校验器的时候，请记得设置 errors 的值。

> Rails 5 里 `valid?` 和 `invalid?` 可加类似 `[:create, :update]` 这样的参数进行多个上下文进行校验。

调用方式比较

	validate	validates 和 validates!	validates_each	validates_with	validates
属性	不需要	需要	需要	不需要	
校验器	不需要	需要	不需要	需要	
直接、间接	直接	间接	直接	间接	

> Note: 为了降低理解难度 validates_each 用到的 BlockValidator 不算为校验器；
> validates_presence_of 代表着其它与之类似的方法。

上面说的都是类级别的校验，如果需要针对某个实例对象单独做额外的校验，可以使用实例方法 validates_with，参数和同名类方法一样。

使用 validate

之前版本可以在 before_save 等方法里，设置 `record.errors.add :xxx` 然后 `return false` 做校验，中断后续操作。

Rails 5 之后默认关掉了这个功能，即不会中断后续操作。推荐使用 `validate :method_name` 在 `method_name` 写对应校验代码。

Validator

校验可以简单分为 3 个层次：1）完全使用原生的；2）使用原生的 + 带条件判断，或自行设置 errors；3）自定义校验器

自定义校验器，有两种方式：继承于 Validator 或 EachValidator.

继承于 Validator

校验器在整个项目生命周期中只初始化一次。它针对的是整个对象，并且自动执行。这种方式，由 `validates_with` 加校验器名字的方式进行调用。

任何继承于 ActiveModel::Validator 的校验器都要实现 `validate` 方法，此方法接收要校验的 record 做为参数。然后，通过 `validates_with` 方法可以使用刚才定义的校验器。

```
class MyValidator < ActiveModel::Validator
  # 继承于 Validator，需要实现 validate
  def validate(record)
    record  # => The person instance being validated
    options # => Any non-standard options passed to validates_wi
th
  end
end

class Person
  include ActiveModel::Validations

  # 调用方式
  validates_with MyValidator
end
```

继承于 EachValidator

实际上，推荐使用这种方式。这种方式，由 `validates` 以参数的方式进行调用。

任何继承于 ActiveModel::EachValidator 的校验器都要实现 `validate_each` 方法，此方法接收要校验的 record、attribute、value 做为参数。

```
class TitleValidator < ActiveModel::EachValidator
  # 继承于 EachValidator，需要实现 validate_each
  def validate_each(record, attribute, value)
    record.errors.add attribute, 'must be Mr., or Mrs.' unless %
w(Mr. Mrs.).include?(value)
  end
end
```

然后，通过 `validates` 方法(刚才定义的校验器做为参数之一)可以使用刚才定义的校验。

```
class Person
  include ActiveModel::Validations
  attr_accessor :name

  # 调用方式
  validates :name, title: true
end
```

语法糖：

```
ActiveRecord::Base.class_eval do
  def self.validates_date_of(*attr_names)
    validates_with TitleValidator, _merge_attributes(attr_names)
  end
end
```

```
class Person < ActiveRecord::Base
  # ...

  # 调用方式
  validates_title_of :name
end
```

> Note: EachValidator 也继承于 Validator.
> 但 Rails 所有所有对外提供的校验器，都继承于 EachValidator.

Errors

有校验就会有失败，对属性校验失败时的报错，是它的实例。

```
record.errors.class
=> ActiveModel::Errors
```

自己写校验方法或者校验器的时候，请务必设置 errors 的值，并且不通过进返回 false.

常用以下方法：

方法	解释
[]	有则 get，无则 set
add	追加错误信息到属性，但在这之前做了一点格式化
keys	返回所有错误的 key
values	返回所有错误的 value
each	可以循环获取每一个错误的 key 和 value
full_messages	返回所有的错误。但已经用 full_message 对它们进行了格式化，方便阅读
blank? & empty?	errors 这个对象是否为空，也就是说是否有错误
details	查看 errors 详情

除以上外，还有方法：

```
added?

size
count

clear
delete

has_key? & include?

to_a
to_hash
to_xml
as_json

full_message
full_messages_for

initialize_dup

generate_message
```

视图里常用写法举例：

```
<%= @record.errors.full_messages.join(";") %>

<%= @record.errors.full_messages.first %>
```

```
<% if @record.errors[:title].present? %>
  <%= @record.errors[:title].join('') %>
<% end %>
```

自己定义的校验举例：

```
# 校验开始工作日期不能早于今天
before_save :validates_of_beginning_work_date

def validates_of_beginning_work_date
  if self.beginning_work_date_changed? && (self.beginning_work_d
ate < Time.now.beginning_of_day)
    # errors 不为空，做提醒。回调返回 false 就不能保存。
    self.errors.add :beginning_work_date, '不能小于当前时间'
    return false
  end
end

# 注意在 local 里对 beginning_work_date 进行翻译，以便更友好的显示报错
信息。
```

> Note: 因为 include Enumerable，所以可以看出很多与其同名的方法。

Validations 相关的 Callbacks

在执行校验之前、之后做某事。

提供 `before_validation`、`after_validation` 这两个和校验有关的回调方法。

```
class MyModel
  include ActiveModel::Validations::Callbacks

  before_validation :do_stuff_before_validation
  after_validation  :do_stuff_after_validation

  def do_stuff_before_validation
    # ...
  end

  def do_stuff_after_validation
    # ...
  end
end
```

内部实现，调用了 Active Support 提供的回调相关方法；使用上，和 Active Record 里的回调方法类似。

Conversion

数据库提供的 record 并不是所有数据对我们都有用，有时候我们需要转换。常见的转换方法有：

```
to_key
to_model
to_param

to_partial_path
```

举例 `to_param` 使用到它的一些方法：

```
url_for

button_to
```

此时，会用 id 做为 url 的一部分，这对用户体验和 SEO 都不太友好，通常我们可以在 model 里覆盖此方法。

```
def to_param
  "#{id}-#{title.parameterize}"
end
```

Naming & Name

内省机制，主要负责将对象转换成对应的字符串。对于我们 Web 开发者来说不常用，但对于配合 Action Controller, Action View 工作很重要。

实例方法：

```
model_name
```

类方法(调用方式奇怪，一般我们不会使用)：

```
param_key
route_key
singular_route_key

singular     # 单数
plural       # 复数
uncountable? # 不可数？
```

方便我们把，字符串、符号、实例对象等转换成相关 model 进行处理。

`model_name` 返回的是 **Name** 的实例对象。Name 和普通字符串类似，但提供方法：

```
human
```

和

```
attr_reader :singular, :plural, :element, :collection,
  :singular_route_key, :route_key, :param_key, :i18n_key,
  :name

alias_method :cache_key, :collection
```

可将字符串转换成对用户更友好的形式展现。

`model_name` 把"各种对象"转换成对应的"字符串"，而 View 要的正是"字符串"。比如以下几个方法：

Action View

```
form_for

button
submit

fields_for

dom_id
dom_class
```

Action Controller

```
wrap_parameters
```

Action Dispatch

```
polymorphic_url
polymorphic_path
```

相关 ActionController::ModelNaming 和 ActionView::ModelNaming.

> Note: 调用此模块的方式是 extend ActiveModel::Naming，不是 include；另外注意实例方法与类方法的调用方式是不一样的。

~~Translation~~

`human_attribute_name`

根据属性名，自动转换成对人类阅读更加友好的字符串。如 "first_name" 转换成 "First name".

使用场景非常非常有限。举例：View 根据字段，自动生成的内容；格式良好的 Errors 报错信息等。

`lookup_ancestors` 找出 respond_to?(:model_name) => true 的祖先。

> Note: 这里的方法用得很少，并且第一个方法里的"属性名"，其实不是属性也可以。

Lint Tests

前面提到 Active Model 最重要的是 Model. 想要去掉 Active Record，使用其它 ORM，基本的兼容性是要做的。如何检测基本的兼容性做到了？通过 Lint::Tests 可以检测即可。

它主要测试一些"必不可少"的基本方法：

```
test_errors_aref
test_model_naming

test_to_key
test_to_param

test_to_partial_path

test_persisted?
```

Model 的增强模块

TODO

Attribute Assignment

以 Hash 的形式给某个对象赋值，并且传递的属性经过 ForbiddenAttributesProtection 检查。

```
assign_attributes
```

是 update & update_attributes 的底层实现。参数的类型都是 Hash 对象，但它不会触发 `save` 操作。

使用举例：

```
# 直接赋值
cat = Cat.new(name: "Gorby", status: "yawning")
cat.attributes # => { "name" => "Gorby", "status" => "yawning",
"created_at" => nil,
                      "updated_at" => nil}

# 使用 Attribute Assignment
cat.assign_attributes(status: "sleeping")
cat.attributes # => { "name" => "Gorby", "status" => "sleeping",
 "created_at" => nil,
                      "updated_at" => nil }
```

原理上它和直接赋值是一样的（用了元编程一个个属性直接赋值），只是对于要传递的参数多了 ForbiddenAttributesProtection 检查。

对比直接赋值：

```
user.is_admin = true
# 直接赋值，不涉及 ForbiddenAttributesProtection

user.assign_attributes is_admin: true
# ActiveModel::MassAssignmentSecurity::Error:
# Can't mass-assign protected attributes: is_admin
```

我们几乎不会直接使用 `assign_attributes` 给对象赋值。

Attribute Methods

Attribute Methods 可以很方便给现有属性(可以是虚拟属性)添加前缀、后缀或前缀 + 后缀，然后得到新的方法。

使用步骤:

1. include ActiveModel::AttributeMethods
2. 调用 `attribute_method_prefix` 添加前缀，调用 `attribute_method_suffix` 添加后缀，调用 `attribute_method_affix` 添加前缀 + 后缀
3. 在这之后，调用 `define_attribute_methods` ，指明对哪些属性有效(默认是所有)
4. 定义一个 `attributes` 方法。返回值是一个 hash，属性的名字做为 key，属性的值做为 value. 实际上可实现其读、写方法，相当于 attr_accessor
5. 用 attribute 加上刚才的前缀、后缀做为方法名，定义一个新的方法

单独使用 Attribute Methods 举例:

```
class Person
  include ActiveModel::AttributeMethods

  # 前缀
  attribute_method_prefix 'clear_'
  # 后缀
  attribute_method_suffix '_contrived?'
  # 前缀 + 后缀
  attribute_method_affix  prefix: 'reset_', suffix: '_to_default
!'

  # 要处理的属性
  define_attribute_methods :name

  attr_accessor :name

  # 供查询属性及值
  def attributes
    { 'name' => @name }
  end

  private
  # 用 attribute 代替上面要处理的属性，加上前缀、后缀得到新的方法名，然后
实现它们

  def attribute_contrived?(attr)
    true
  end

  def clear_attribute(attr)
    send("#{attr}=", nil)
  end

  def reset_attribute_to_default!(attr)
    send("#{attr}=", 'Default Name')
  end
end
```

Rails 项目，有的步骤默认已经实现，所以使用上可以减少几个步骤，但其中的第 2，5 步是无论如何都不可省。

```
class Person < ActiveRecord::Base
  attribute_method_affix prefix: 'me_mateys_', suffix: '_is_in_p
irate?'

  private

  def me_mateys_attribute_is_in_pirate?(attr)
    send(attr).to_s =~ /\bYAR\b/i
  end
end

person = Person.find(1)
person.name                               # => 'Paul Gillard'
person.profession                         # => 'A Pirate, yar!'
person.me_mateys_name_is_in_pirate?       # => false
person.me_mateys_profession_is_in_pirate? # => true
```

主要用到的方法，都在示例里。Active Record 基于它也实现了一个自己的 Attribute Methods.

类方法

```
attribute_method_prefix
attribute_method_suffix
attribute_method_affix

alias_attribute  # 起别名
attribute_alias? # 起别名了？
attribute_alias  # 原名字是...

define_attribute_methods
define_attribute_method
undefine_attribute_methods

generated_attribute_methods
```

> Note: ActiveRecord::AttributeMethods include ActiveModel::AttributeMethods 并修改了它的部分函数，所以使用 Rails 的话，这些方法是用不了的。

Dirty

跟踪对象的变化情况。其实，它们都属于脏数据，所以起这名字。但有时候很有用，例如某个字段一经生成不允许更改，或者某字段每次更改要确保与上次不同。

使用 ActiveModel::Dirty，需要:

- include ActiveModel::Dirty
- 调用 `define_attribute_methods` 参数是你想跟踪的属性
- 在改变属性的值之前，调用 `attr_name_will_change!` (把 attr_name 换成真正的属性名)
- 在改变属性的值之后，调用 `changes_applied`

举例:

```
class Person
  include ActiveModel::Dirty

  define_attribute_methods :name

  def name
    @name
  end

  def name=(val)
    name_will_change! unless val == @name
    @name = val
  end

  def save
    # do persistence work
    changes_applied
  end

  def reload!
    # reset_changes
  end
end
```

运用 **Attribute Methods** 生成的方法，能精确跟踪到某属性：

```
attribute_method_suffix '_changed?', '_change', '_will_change!',
'_was'

attribute_method_affix prefix: 'restore_', suffix: '!'
```

下面是对它们的解释

```
x_changed?      # 返回 true/false, x 属性有没有更改？
x_change        # 返回一个数组有两个元素，第 1 个为 x 属性更改前的值，第
2 个为 x 属性更改后的值
x_will_change! # 声明 x 元素已被更改，即使实际上它并没有更改。
x_was           # 根据 x_changed? 而来。
                # 如果有更改则返回 changed_attributes 里 x 属性的部分
，没有更改则返回 x 属性的值

restore_x!      # 消除对 x 属性的更改
```

除上述方法外，还有

```
changed?        # 返回 true/false，整个对象有没有被更改？
changed         # 返回一个数组，所有被更改的属性
changes         # 返回一个 Hash. key 被更改的元素，value 是一个数组
                # (有两个元素，第 1 个为 x 属性更改前的值，第 2 个为 x 属性更改后的值)
previous_changes    # 类似 changes, 区别是更新成功之后才使用。
                    # 返回一个 Hash. key 被更改的元素，value 是一个数组

                    # (有两个元素，第 1 个为 x 属性更改前的值，第 2 个为 x 属性更改后的值)
changed_attributes # 返回一个 Hash. key 为被更改的元素，value 为其更改之前的值
restore_attributes # 清除更改数据

changes_applied
clear_changes_information

attribute_previously_changed? # 对已经保存的对象，之前改变了哪些属性
attribute_previous_change # 改变的属性值是什么
```

Active Record 也有同名模块，它只是对这里的 Dirty 的封装，并且它并没有对外提供 API.

Secure Password

类方法：

```
has_secure_password(options = {})
```

实例方法：

```
authenticate(unencrypted_password)

attr_reader :password

# 以下两方法和 attr_accessor 类似
password=(unencrypted_password)
password_confirmation=(unencrypted_password)
```

依赖于 `gem 'bcrypt'`，必须有 `password_digest` 属性(可以没有 `password` 属性)，使用参考：

```
# Schema: User(name:string, password_digest:string)
class User < ActiveRecord::Base
  has_secure_password
end

user = User.new(name: 'david', password: '', password_confirmati
on: 'nomatch')
user.save                                               # => false, 密码
不能为空
user.password = 'mUc3m00RsqyRe'
user.save                                               # => false, 确认
密码失败
user.password_confirmation = 'mUc3m00RsqyRe'
user.save
# => true
user.authenticate('notright')
# => false
user.authenticate('mUc3m00RsqyRe')
# => user
User.find_by(name: 'david').try(:authenticate, 'notright')
# => false
User.find_by(name: 'david').try(:authenticate, 'mUc3m00RsqyRe')
# => user
```

使用 has_secure_password 后，还会自动帮我们添加校验：

```
validates_length_of       :password
validates_confirmation_of :password
```

下面是 Rails 里面默认的加密、解密实现：

```
require 'bcrypt'
# => true

cost = BCrypt::Engine.cost
# => 10

unencrypted_password = "password"
# => "password"

# 加密
password_digest = BCrypt::Password.create(unencrypted_password,
cost: cost)
# => "$2a$10$GGtvADq0jfb9E2wy4Nq0je1ZrJbJrsRSigwtBMlAAfV5dbAEgjt
7C"

# 解密
BCrypt::Password.new(password_digest) == unencrypted_password
# => true
```

Forbidden Attributes Protection

Strong Parameters 默认是黑名单，params 在(controller层面) permit 后状态会变成 permitted (进入白名单)，只有 permit 的属性才能进入我们系统。为了防止程序员遗漏或图省事，直接传递白名单，我们可以在(model层面)里做校验，如果发现有属性不在白名单内，则报错：

```
# app/controllers/people_controller.rb
class PeopleController < ActionController::Base
  def create
    # 正常情况
    Person.create(person_params)

    # 遗漏或图省事
    Person.create(params[:person])
  end

  private

    def person_params
      params.require(:person).permit(:name, :age)
    end
end
```

```
# app/models/people.rb
class Person < ActiveRecord::Base
  include ActiveModel::ForbiddenAttributesProtection
end
```

此模块用于遗留项目，它的使用，还需要安装 gem 'strong_parameters'

```
# 原理
if attributes.respond_to?(:permitted?) && !attributes.permitted?
  raise ActiveModel::ForbiddenAttributesError
end
```

其它

Rails 的 where 查询也有类似 Mass Assignment 保护。

它主要针对的是：在 Controller 和 View 里通过 `params` 给 record 对象属性赋值 (AttributeAssignment)

注意，这里做了判断，并不是强制执行，所以和 ActionController 算不上强耦合。

Serialization

`serializable_hash(options = nil)` 序列化操作，提供了 :only, :except 选项。常用的 `as_json` 封装并扩展了它。

使用 Serialization 需要 `record.respond_to? :attributes` # => true 因为转换的过程使用到相应 model 的 attributes 实例方法。

此外，还有相关 module 及方法：

JSON

```
# 封装了上面的 serializable_hash，并提供 :root 选项
as_json

# 调用了 attributes= 方法，给对象赋值
from_json
```

~~Xml~~

```
# 将对象转化成 xml 格式
to_xml

# 先将 xml 格式成 Hash，再调用了 attributes= 方法，给对象赋值
from_xml
```

> 该模块在 Rails 5 已被移除。

注意几个转化成 **json** 方法之间的区别：

```
person = Person.new
person.name = "Bob"

person.serializable_hash # => {"name"=>"Bob"}
person.as_json           # => {"name"=>"Bob"}
person.to_json           # => "{\"name\":\"Bob\"}"
```

Callbacks - 快速提供 3 个回调方法

`define_model_callbacks(*callbacks)` 快速定义 Model 里使用的 3 个回调方法，它们是：before_x、around_x 和 after_x.

一般的，ORM 都会提供大量的回调函数，其中有的彼此之间还很相似。直接使用 Active Support 那一套回调机制的话，略显麻烦，还会有重复代码。所以，Active Model 封装了 Active Support，使得定义回调函数变得简单方便。

使用举例：

```
require 'active_model'

class Person
  extend ActiveModel::Callbacks

  define_model_callbacks :update

  # 运用 define_model_callbacks 自动生成的方法
  before_update :reset_me
  # around_update ...
  after_update  :say_success

  # 为了统一，易于理解，这里的方法名一般用：update
  # 但这并不是规定死的，也可以不一致。比如这里用的是：update_me
  def update_me
    run_callbacks(:update) do
      p "update_me 真正要执行的代码"
      p "... before、around 和 after 就是针对这里的代码而言的！"
    end

    p "update_me 真正要执行的代码，还可以有其它内容"
    p "... 但为了简单、避免混淆，不建议再放其它任何代码！"
  end

  def reset_me
    p "在 update_me 真正要执行的代码之前，执行这里的代码"
  end

  def say_success
    p "在 update_me 真正要执行的代码之后，执行这里的代码"
  end
end

person = Person.new
person.update_me
```

上面例子里的 `before_update` 和 `after_update` 方法我们并没有显式定义，是 define_model_callbacks 帮我们自动生成的。

其它

~~Forbidden Attributes Error~~

```
class Person < ActiveRecord::Base
end

params = ActionController::Parameters.new(name: 'Bob')
Person.new(params)
# => ActiveModel::ForbiddenAttributesError

params.permit!
Person.new(params)
# => #<Person id: nil, name: "Bob">
```

Active Record 数据库增删查改

Web 应用使用到数据库，而管理数据库使用的是 SQL 语言。我们不需要专门去学习 SQL，只需要用 Ruby 语言，写 Ruby 代码就能实现数据库的相关操作(也就是各种简单、复杂的读写操作)。

- Scoping

包括 Default、Named

- Relation

包括 Spawn Methods、Query Methods、Finder Methods、Calculations、Batches、~~Delegation~~、~~Predicate Builder~~、~~Merger~~

- Querying

- Persistence

即将被废除的 gem 'activerecord-deprecated_finders'

- Counter Cache

- Attribute Assignment

主要工作由 ActiveModel::AttributeAssignment 完成，这里不再赘述。

- Attribute Methods

包括 Before Type Cast、Dirty、Primary Key、Query、Read & Write、Serialization、Time Zone Conversion

- ~~Null Relation~~

- ~~Dynamic Matchers~~

Relation(Arel)

对 Relation 的操作。哦，我们先理解概念。

- 为什么不完全用 Ruby 或 SQL？

Ruby 慢，人性化；SQL 快，不易读写。

- 如何充分利用两者优点，特别是我们要做复杂查询的时候？(复杂查询 = 简单的查询 + 简单的查询 + ...)

"(Ruby 查询 -> SQL 查询，Ruby 查询 -> SQL 查询) -> 结果" 比 "(Ruby 查询 + Ruby 查询 -> SQL 查询) -> 结果" 效率要低。所以尽量不要在 Ruby - SQL 两个层面之间转来转去，尽量把多个 Ruby 查询转化成相应的一个 SQL 查询。

- 如何实现？

运用中间状态，也就是 Relation. 每次查询并不是真正的查询(因为没走到 SQL 层面)，而是保存一个中间状态，当你所有的查询条件都写完了，才进入 SQL 的层面。理论上，这些简单的查询最后都能组合成 SQL 语句。

- 好处？

延迟加载。我们在 Controller 里有一个查询语句，结果赋值给一个实例变量，原本的意图是在 View 里显示的。某天需求更改了，我们不必再显示这个查询结果，但 Controller 里我们忘记删除这部分的代码(这都能忘？)，结果每次都要做大量无用的查询工作。引入中间状态后，就能起到延迟加载的作用，不到最后的调用，不做 SQL 查询(停留在 Ruby 层面)。

链式查询，效率高。上面已经提到了"(Ruby 查询 -> SQL 查询，Ruby 查询 -> SQL 查询) -> 结果" 比 "(Ruby 查询 + Ruby 查询 -> SQL 查询) -> 结果" 效率要低。

- 如何区分？

在控制台里执行一下查询命令，看返回结果的 class (类型)。有 ActiveRecord::Relation、 ActiveRecord::AssociationRelation 和 ClassName::ActiveRecord_Relation 等包含 Relation 字样的就是了。

Relation 就类似没有名字的 scope 。当涉及跨表查询时，使用链式查询可以很大程度的提高效率。更多请查看接口 Active Record Query Interface

> Note: 这里部分是对多个对象的操作，对 Relation 的操作；不是查询操作。

Relation 文件下的方法

方法	解释
build & new	新建 record 对象
create	新建并保存 record 对象
create!	新建并保存 record 对象，如果有异常则报错
delete	基于 SQL 删除指定 id 的 record 对象
delete_all	基于 SQL 删除 relation 包含的所有 record 对象 可以先把这些 record 对象查询出来，也可以删除的时候传条件进行查询
destroy	删除指定 id 的 record 对象，会运行回调函数
destroy_all	删除 relation 包含的所有 record 对象，会运行回调函数
find_or_create_by	根据所给的属性，查询 record 对象；若查询不到，则创建
find_or_create_by!	根据所给的属性，查询 record 对象；若查询不到，则创建。创建则有可能失败，失败报错
find_or_initialize_by	
find_or_create_by!	根据所给的属性，查询 record 对象；若查询不到，则初始化
update	传递参数 ids、attributes，所有指定 id 的 record 对象都会更新 用的是自己的 update 方法
update_all	更新整个 relation 包含的 record 对象(先查询出来)，参数为更新内容

和

方法	解释
==	判断集合是否等价于 relation. 这里的集合，并不限于 relation，也可以数组
any?	非空？
empty?	为空？
blank?	为空？
many?	多个？ 也就是 relation.count > 1
none?	...
one?	...
explain	1. 模拟在各自数据下的执行效果 2. 解释生成的 sql
reset	如果对所得的 relation 进行了处理，但还没有保存，使用它可重置之前的处理
load	执行查询操作，但返回的是 relation
reload	reset + load
cache_key	...
values	返回规格化的所有查询条件。如果你用的查询条件太多了，可用它来查看
where_values_hash	返回所有 where 查询条件，结果并不准确
to_sql	一般地，使用 `to_sql` 可以方便的查看生成的 Sql 语句
size	...

除上述方法外，还有：

```
to_a

eager_loading?

encode_with

initialize_copy

inspect

joined_includes_values

pretty_print

scope_for_create

scoping

uniq_value
```

我们不需要也不推荐使用 `to_a` 强制转换 Relation 成数组。

Query Methods

介绍的方法都在 ActiveRecord::QueryMethods

提供方法：

方法	解释
bound_attributes	
create_with	创建一个 record 对象，调用者是 Relation 对象. 没找到合适的使用场景 设置的是 create_with_value 的值，在很多地方会间接用到它
distinct & uniq	通常要配合其它查询方法使用，返回是 Relation. 否则使用的是数组的 uniq 返回的不是 Relation
eager_load	后文解释
extending	给一个 scope 增加方法，返回的仍然是 scope. 如果传递的是 block, 则可以直接调用 block 里面的方法, 如果传递的是 module, 则可以调用 module 里面的方法。 不推荐直接使用，这会大大提高复杂度，但扩展时可以使用。
from	从符合条件的 record 开始
group	后文解释
having	having 是分组(group)后的筛选条件，分组后的数据组内再筛选; where 则是在分组前筛选
includes	后文解释
joins	后文解释
left_joins & left_outer_joins	
limit	限制结果数目
lock	锁定结果
none	返回一个空的 Relation
offset	类似 from, 但它传递的是"第几条"，并且数据不够的话可循环；而后者传递的是查询条件，不符合条件返回空
or	对应 SQL 里的“或”查询

preload	后文解释
readonly	查询结果只读，不可写
references	后文解释
reorder	重新按条件排序。在这之前有排序的话，忽略它们
reverse_order	反转之前的排序结果
rewhere	重新按条件查询。在这之前的排序条件和它没有冲突的情况下，保留它们；有冲突的情况下，忽略它们
select	后文解释
unscope	查询条件可以有多个。使用 unscope 可以忽略其中的一个或多个
where	后文解释

`where(opts = :chain, *rest)`

- 传递字符串。

不过这容易引起注入攻击。

- 传递数组。

1) 数组第一个元素做为查询条件(占位符'?')，后面的元素按顺序辅助完成；
2) 数组第一个元素做为查询条件(占位符':key')，后面的元素放到一个哈希里，辅助完成；
3) 数组第一个元素做为查询条件(占位符'%')，后面的元素转义后，按顺序辅助完成。

传递多个参数时，默认把它们当做数组处理，尽管没有使用中括号 []

- 传递哈希。

包括用花括号 {} 或者裸哈希。

使用哈希，还有一个好处，可以很直接的使用范围，如：

```
Comment.where(created_at: @the_date.beginning_of_day..@the_date.
end_of_day)
```

- 配合 joins

如果要 joins 其它表，where 里的查询语句参数和之前一样，但被 join 的表名要包含在查询条件里。

```
User.joins(:posts).where({ "posts.published" => true })
User.joins(:posts).where({ posts: { published: true } })
```

`select(*fields)`

默认 find 查询所有属性，也就是 select *.

如果只想查询部分属性，你可以自己指定：

- 传递 block，返回数组
- 传递属性名(列)，结果仅包含这些项

`group(*args)`

根据所传递的属性，对结果进行分组。和 select 对比：

```
User.select([:id, :name])
=> [#<User id: 1, name: "Bar">, #<User id: 2, name: "Bar">,
    #<User id: 3, name: "Foo">

User.group(:name)
=> [#<User id: 3, name: "Foo", ...>, #<User id: 2, name: "Bar",
...>]

User.where('tag_list != ?', '').group(:tag_list).order('count_al
l desc').count
```

group 通常要配合其它查询语句或条件使用，单独使用只会有"去重"效果(record 属性相同的话，只保留最后一个 record 对象)。

`having(opts, *rest)`

和 group 对比：having 是筛选条件，group 是分组。

```
Rails 里 having 不能脱离 group 单独使用。

Order.having('SUM(price) > 30').group('user_id')
```

`order(*args)`

需要知道：属性和规则。如果不指定规则，则用默认的。

`limit(value)` 和 `offset(value)`

指定最多获取多少条数据。

```
User.limit(10) # generated SQL has 'LIMIT 10'
User.limit(10).limit(20) # generated SQL has 'LIMIT 20'
```

指定从第几条开始获取数据。

类比：分页时的'第几页'类似 offset, 而'每页多少条数据'类似 limit

`readonly(value = true)`

注意是数据库返回的结果就是只读，和 `attr_readonly` 是两码事。

```
users = User.readonly
users.first.save
=> ActiveRecord::ReadOnlyRecord: ActiveRecord::ReadOnlyRecord
```

`reorder(*args)`

重新排序。调用了其他人写的 query 方法，但对排序不满意的，可以重新排序。

`joins(*args)`

如果要根据关联表来查询，joins 后面的 where 查询语句里要包含被 join 表的表名。

```
Category.joins(:posts)
# => SELECT categories.* FROM categories
  INNER JOIN posts ON posts.category_id = categories.id

Post.joins(:category, :comments)
# => SELECT posts.* FROM posts
  INNER JOIN categories ON posts.category_id = categories.id
  INNER JOIN comments ON comments.post_id = posts.id

product = Product.first
product.catalogs.joins(:catalogs_products)
  .where('catalogs_products.is_default = ?', true)
# => SELECT DISTINCT `catalogs`.* FROM `catalogs` INNER JOIN `ca
talogs_products`
     `catalogs_products_catalogs` ON
     `catalogs_products_catalogs`.`catalog_id` =
     `catalogs`.`id` INNER JOIN `catalogs_products` ON `catalogs`
.`id` =
     `catalogs_products`.`catalog_id` WHERE
     `catalogs_products`.`standard_product_id` = 100011 AND
     (catalogs_products.is_default = 1)
```

`includes(*args)` 和 joins 类似，唯一的区别在于：它是预先加载，第一次执行会慢，后续不再执行。

```
Post.includes(:comments).where("comments.visible" => true)
```

`distinct(value = true)` aliased as: uniq

`explain()` 看看转化成 SQL 是什么样。

`none` 返回一个空的 Relation，对后续操作很有用。可以充分利用 Relation 链式调用、延迟加载等特性。

`not` "与或非"里面的"非"，例如"IS NOT NULL"查询。

`references` 使用 includes 可以预加载多个关联表，如果后续还有根据关联表进行查询的，需要用 references 指明用哪张关联表。否则，查询会出错。但如果后续的查询是以 hash 的形式提供的话，则不必使用 references 也可以。

`order` 排序。它在 default_scope 之前执行，也就是说 default_scope 有可能会覆盖之前的排序。

`left_outer_joins` 使用举例：

```
User.left_outer_joins(:posts)
# => SELECT "users".* FROM "users" LEFT OUTER JOIN "posts" ON
     "posts"."user_id" = "users"."id"
```

> Note: 注意区分 Rails 里的 group 和 SQL 里的 group_by

相关 SQL SQL Functions
参考官方文档 Active Record Query Interface

最后

查表操作(数据库读操作)。大部分是 Ruby 层面，一般可多条件链式查询。

Query Methods 里的方法和 Collection Proxy 里的方法很类似，区别在于前者主要是对当前表进行操作；而后者主要是对其关系表进行操作。

Query Methods 和 Finder Methods 也有类似之处，区别在于前者返回的是 Relation 对象，可以链式查询；而后者直接返回的已经是结果，不可再链式查询。

另，注意：参数带 block 的，会离开 Ruby 层面，执行 SQL 查询，返回结果。

要善于使用这里的语句，数据放在数据库和数据放在内存有很大的区别；放在内存和是否返回也有很大的区别。

> Note: 返回的多是 Relation，与 SQL 层面较亲；有 find 字样的绝对不是它。

Preload, Eagerload, Includes 和 Joins 等

延迟加载，如 Relation，scope 预先加载，如 includes

N + 1

```
# 一次查询
doctors = Physician.all

# N 次查询
doctor.patients.each do |patient|
  puts patient.name
end
```

includes

把关系表数据也查询出来。两个查询都要做，关联对象也需要放到内存。

```
User.includes(:posts)

=> SELECT "users".* FROM "users"
=> SELECT "posts".* FROM "posts"  WHERE "posts"."user_id" IN (1,
2, 3)
```

```
# 第一次调用，需要查询，花销大
# 目的：查询主表和关联表
User.includes(:posts).where('posts.desc = "ruby is awesome"').re
ferences(:posts)

# =>
SELECT "users"."id" AS t0_r0, "users"."name" AS t0_r1, "posts"."
id" AS t1_r0,
       "posts"."title" AS t1_r1,
       "posts"."user_id" AS t1_r2, "posts"."desc" AS t1_r3
FROM "users" LEFT OUTER JOIN "posts" ON "posts"."user_id" = "use
rs"."id"
WHERE (posts.desc = "ruby is awesome")

# 再次调用，不需要查询，花销为零
@customers = Customer.joins(:products).where("products.is_master
 = true")
```

特点，生成一条还是两条 SQL 查询语句，取决于写法。

可分为 2 种不同情况，和 preload 一样、和 earge_load 一样。

性能最慢。

通过中间表的话，默认也加载。

`a.includes(:bs).where(bs.x ...)` includes 只包含符合条件的 a 和 a 下面符合条件的 bs

> includes 如果不接 where 查询，则只做预加载，没有过滤的作用。

joins

普通的查询条件，关联对象不会放到内存。

```
User.joins(:posts)
=> SELECT "users".* FROM "users" INNER JOIN "posts"
   ON "posts"."user_id" = "users"."id"

# 第一次调用，需要查询，花销一般
# 目的：查询主表，关联表仅做为查询条件之一

posts = Post.joins(:comments)
# => Post Load (0.1ms)  SELECT "posts".* FROM "posts" INNER JOIN
     "comments" ON "comments"."commentable_id" = "posts"."id" AN
D
     "comments"."commentable_type" = 'Post'

# 再次调用，也需要查询，花销一般
posts.first.comments
# => Comment Load (0.2ms) SELECT "comments".* FROM "comments"
     WHERE "comments"."commentable_id" = ? AND "comments"."comme
ntable_type" = ?
     [["commentable_id", 1], ["commentable_type", "Post"]]
```

特点，不会查询出关联表的数据，仅做为查询条件。

> joins 即使不接 where 查询，也有过滤作用，默认是 INNER JOIN.

复杂的 joins

```
# has_and_belongs_to_many
product has_and_belongs_to_many :devices

Product.joins("join devices_products on products.id = devices_products.product_id")
       .where(["devices_products.device_id = ?", params[:device_id]])

# has_many :through
has_many :catalogs_products
has_many :catalogs, :through => :catalogs_products

Product.joins(:catalogs_products).where(catalogs_products:
                                        {catalog_id: params[:catalog_id]})
```

preload

类似 includes 的子集。

```
User.preload(:addresses)

=> SELECT "users".* FROM "users"
=> SELECT "addresses".* FROM "addresses"  WHERE "addresses"."user_id" IN (1, 2)
```

特点，SQL 查询语句始终是两条(单独的数据库查询)。

后续，不能以关联表做为查询条件。

性能最快。

通过中间表的话，默认也加载。

`a.preload(:bs).where(bs.x ...)` preload 包含符合条件的 a 和 a 下面所有的 bs

> Note: 实际应该是 a.joins(:bs).where(bs.x ...).preload(:bs)

eager_load

```
User.eager_load(:posts)
=> SELECT "users"."id" AS t0_r0, "users"."name" AS t0_r1, ...
FROM "users" LEFT OUTER JOIN "posts" ON "posts"."user_id" =
"users"."id"
```

特点，SQL 查询语句始终只有一条(使用了 LEFT JOIN)。

性能中等。

default_scope 不起作用。

通过中间表的话，要明确指出才加载。

references

includes 后面的查询条件，用的是 "关联表.属性"，有时候 Rails 不能推断出这个'关联表'到底是哪个(可以 includes 多个关联表)，需要用 references 指明。

Rails 3 比较'宽松'，includes 有时候不指定 references 也可以工作。 Rails 4 比较'严格'，includes 需要指定 references 才能工作。

举例：

```
User.includes(:posts).where("posts.name = 'foo'")
# => Doesn't JOIN the posts table, resulting in an error.

User.includes(:posts).where("posts.name = 'foo'").references(:po
sts)
# => Query now knows the string references posts, so adds a JOIN
```

参考

3 ways to do eager loading (preloading) in Rails 3 & 4
eager loading in rails

Query Methods 读取、设置查询方法

除【**Query Methods**】介绍的方法外，还有以下读取、设置查询方法。它们都是用元编程生成的。

在 Relation 文件下，有代码：

```
MULTI_VALUE_METHODS  = [:includes, :eager_load, :preload, :select
, :group,
                        :order, :joins, :where, :having, :bind,
:references,
                        :extending, :unscope]

SINGLE_VALUE_METHODS = [:limit, :offset, :lock, :readonly, :from
, :reordering,
                        :reverse_order, :distinct, :create_with,
:uniq]
```

在 Query Methods 文件下，使用它们。读取或设置指定的查询条件。

```
MULTI_VALUE_METHODS.each do |name|
  def #{name}_values                # def select_values
    @values[:#{name}] || []         #   @values[:select] || []
  end                               # end
                                    #
  def #{name}_values=(values)       # def select_values=(values)
    @values[:#{name}] = values      #   @values[:select] = values

  end                               # end
end

(SINGLE_VALUE_METHODS - [:create_with]).each do |name|
  def #{name}_value                 # def readonly_value
    @values[:#{name}]               #   @values[:readonly]
  end                               # end
end

SINGLE_VALUE_METHODS.each do |name|
  def #{name}_value=(value)         # def readonly_value=(value)
    @values[:#{name}] = value       #   @values[:readonly] = val
ue
  end                               # end
end
```

根据以上代码，生成方法：

```
includes_values
includes_values=

eager_load_values
eager_load_values=

preload_values
preload_values=

select_values
select_values=

group_values
group_values=

order_values
order_values=

joins_values
joins_values=

where_values
where_values=

having_values
having_values=

bind_values
bind_values=

references_values
references_values=

extending_values
extending_values=

unscope_values
unscope_values=
```

和

```
limit_value
limit_value=

offset_value
offset_value=

lock_value
lock_value=

readonly_value
readonly_value=

from_value
from_value=

reordering_value
reordering_value=

reverse_order_value
reverse_order_value=

distinct_value
distinct_value=

create_with_value=

uniq_value
uniq_value=
```

关联表复杂查询示例

单群、复数？

关键：

includes、joins 查询对应的是关联名字，where 对应的是表名。

一对多关系时：

joins + where 查询会对每个关联进行查询，所以结果会有重复。
使用 uniq 后前者数目和使用 includes 一样。

includes + where 查询到符合条件就自动终止，所以结果没有重复。
相当于自带 uniq 功能。

一对一关系（**has_one** 或 **belongs_to**）

没有上述差异，因为它在另一个层面解决了重复问题。

判断 nil

```
class Person
  has_many :friends
end

class Friend
  belongs_to :person
end
```

对应：

```
Person.includes(:friends).where( :friends => { :person_id => nil } )
```

```
class Person
  has_many :contacts
  has_many :friends, :through => :contacts, :uniq => true
end

class Friend
  has_many :contacts
  has_many :people, :through => :contacts, :uniq => true
end

class Contact
  belongs_to :friend
  belongs_to :person
end
```

对应：

```
Person.includes(:contacts).where( :contacts => { :person_id => nil } )
```

另外一种解法：

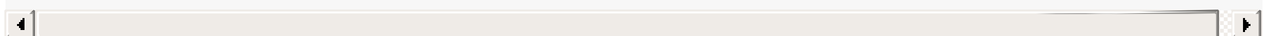
```
Person.includes(:contacts).where( :contacts => { :id => nil } )
```

还有一种丑陋、低效的办法：

```
# 取出所有（去重），不在里面则表示 nil
Person.where('id NOT IN (SELECT DISTINCT(person_id) FROM friends)')
```

如果把 has_many 改为 has_one 呢？

```
Person.includes(:contact).where( :contacts => { :person_id => nil } )
```



判断非 nil

以下几个查询几乎等价：

```
members.includes(:responses).where('responses.id IS NOT NULL')

members.includes(:responses).where.not(responses: { id: nil })

members.joins(:responses)
```

看情况决定是否要加 uniq 条件。

关联的关联

```
class Employee < ActiveRecord::Base
  belongs_to :company
end

class Company < ActiveRecord::Base
  has_many :employees
  has_many :addresses
end

class Address < ActiveRecord::Base
  belongs_top :company
end
```

```
Employee.joins(:company => :addresses).
  where(:addresses => { :city => 'Porto Alegre' })
```

关联对象的数量

一对一

直接上是不会发生的，但实际上，如果没有删除关系对象的话，也会有：

group + having

```
Order.joins( :request_refund ).group( 'orders.id' ).having( 'count( order_id ) > 1' )
```

一对多

group + having

```
Order.joins( :request_refunds ).group( 'orders.id' ).having( 'count( order_id ) > 1' )
```

counter_cache

```
belongs_to :project, counter_cache: true
```

当使用了 `counter_cache` 时可以先运行 `reset_counters` 确保数据准确，然后运用计数器：

```
Project.where( 'vacancies_count > ?', 1 )
```

统计并排序

```
users = User.where('tag_list != ?', '').group(:tag_list).order('count_all desc').count
```

按指定的数组顺序排列

```
Model.where(id: ids).order("field(id, #{ids.joins(',')})")
```

Spawn Methods

方法	解释
except	查询方法有多个，并且可以链式调用。使用 except 忽略之前的某个查询方法
merge	参数是 Relation, 则做为查询条件，返回仍然是 Relation；参数是数组, 则返回前者的查询结果和此数组的交集
only	查询方法有多个，并且可以链式调用。使用 only 指定只能使用的查询方法

当参数是 Relation 时，merge 也和 joins、includes 等一样有联合查询的效果。

`except` 和 `unscope` 功能上类似。区别在于后者在使用上，可以选择更多类型。

除上述方法外，还有：

```
spawn

merge!
```

```
VALID_FIND_OPTIONS = [ :conditions, :include, :joins, :limit, :offset, :extend,
                       :order, :select, :readonly, :group, :having, :from, :lock ]
```

一种 **merge** 转普通查询：

之前

```
User.joins(:account).merge(Account.where(:active => true))
```

之后

```
User.joins(:account).where(:accounts => { :active => true })
```

并不是所有 merge 都可转换成 where，但能转换的话，请务必优先用 where.

Batches

```
find_each

find_in_batches

in_batches
```

一般， `find_each` 后面参数都是 block，此时作用的 `find_in_batches` 类似 (:batch_size 默认是 1000)

Finder Methods

实例方法：

```
exists?

find
find_by
find_by!

take
take!
```

和：

```
first
first!

second
second!

third
third!

fourth
fourth!

fifth
fifth!

forty_two
forty_two!

last
last!
```

除上述方法外，还有一些 protected 方法，但它们一般不会直接使用到。

> Rails 5 里 `find` 可以传递数组，如果没有明确排序的话，默认就使用数组里元素的顺序，这个太赞了。

Calculations

按条件对数据进行统计。

```
count   # 总数
average # 平均值
maximum # 最大值
minimum # 最小值
sum     # 值的总和

pluck # 获取所有指定的属性
ids   # 获取所有的 id 属性

# count, sum, average, minimum, 和 maximum 都是封装它而来
calculate
```

它们都是直接返回结果。

其它

~~Finder Methods & Batches 补充~~

查表操作(数据库读操作)。大部分是SQL层面，一般不可多条件链式查询。

返回的都是结果，不是 Relation。

~~HashMerger & Merger~~

是 Spawn Methods 里的 merge、merge! 方法的底层实现。

在 merger 对象非 Array、非 Relation、非 proc 等情况下才使用到。

~~Delegation~~

~~Predicate Builder~~

Collection Proxy

通过某个对象，对其关联的对象进行操作。
它继承于 Relation，只是稍微做封装，所以接口上也有很多类似之处。

对外提供接口。

有破坏行为：

```
<< & push & append
prepend

to_a & to_ary

reset

replace

reload

create
create!

delete
delete_all

destroy
destroy_all

clear
```

仅是结果：

```
count

any?
many?
empty?

length
size

include?

loaded?

==
```

查询：

```
new & build

concat

distinct & uniq

find

first
last
second
third
fourth
fifth
forty_two

scope & spawn

scoping

select

take
```

其它：

```
arel

target

load_target

proxy_association
```

Scoping

虽然只有 4 个方法，但很实用。

方法	解释
scope	命名 scope
default_scope	设置默认 scope

和

方法	解释
unscoped	跳过之前设置的 scope
all	all 方法，默认已经 scope

scope

重点说说这个方法。

两个参数：第一个是名字，第二个是内容，需要以 proc 的形式定义。

scope 相当于类方法，可检索、查询对象

可执行一系列的查询语句，如：

```
where(color: :red).select('shirts.*').includes(:washing_instruct
ions)
```

```
class Shirt < ActiveRecord::Base
  scope :red, -> { where(color: 'red') }
  scope :dry_clean_only, -> { joins(:washing_instructions)
                              .where('washing_instructions.dry_c
lean_only = ?', true) }
end
```

可以调用 `Shirt.red` 和 `Shirt.dry_clean_only` . `Shirt.red` 功能和 `Shirt.where(color: 'red')` 一样。

也就是说，功能上和以下代码一样:

```
class Shirt < ActiveRecord::Base
  def self.red
    where(color: 'red')
  end
end
```

scope 返回的是 Relation，而不是数组

你可以调用 `Shirt.red.first` , `Shirt.red.count` , `Shirt.red.where(size: 'small')` 等. 但是 Relation 也可以有数组的行为, 如 `Shirt.red.each(&block)` , `Shirt.red.first` , 和 `Shirt.red.inject(memo, &block)` 等。

scope 可以链式调用

`Shirt.red.dry_clean_only` 运行结果是 `red` 和 `dry_clean_only` 综合的结果。
`Shirt.red.dry_clean_only.count` 返回的是 `red` 和 `dry_clean_only` 综合结果的数目。在这里和 `Shirt.red.dry_clean_only.average(:thread_count)` 类似。

scope 是一步步走下去的

```
class Person < ActiveRecord::Base
  has_many :shirts
end
```

假设，elton 是 Person 的实例对象，则 elton.shirts.red.dry_clean_only 返回的是 elton(限制条件) 的 red 和 dry_clean_only shirts.

scope 后可直接跟 extensions

和 has_many 类似的：

```
class Shirt < ActiveRecord::Base
  scope :red, -> { where(color: 'red') } do
    def dom_id
      'red_shirts'
    end
  end
end
```

scope 后可直接跟 creating/building 等方法

用于创建 record

```
class Article < ActiveRecord::Base
  scope :published, -> { where(published: true) }
end

Article.published.new.published    # => true
Article.published.create.published # => true
```

scope 后可直接跟类方法

定义如下：

```
class Article < ActiveRecord::Base
  scope :published, -> { where(published: true) }
  scope :featured, -> { where(featured: true) }

  def self.latest_article
    order('published_at desc').first
  end

  def self.titles
    pluck(:title)
  end
end
```

调用如下:

```
Article.published.featured.latest_article
Article.featured.titles
```

default_scope

1) 一个参数，需要以 proc 的形式定义：

```
class Article < ActiveRecord::Base
  default_scope { where(published: true) }
end
```

2) 始终起作用，不能覆盖，冲突时取合集。

3) 会影响 initialization 过程。例如以上示例，默认新创建的对象 published 为 true.

4) 基于第2、3点请慎用。

unscoped

前面说过 scope 可以链式调用，但如果调用了 `unscoped` 的话，它可以把之前的 scope 给清除掉，包括 default_scope.

示例：

```
class Post < ActiveRecord::Base
  def self.default_scope
    where published: true
  end
end

Post.all
# Fires "SELECT * FROM posts WHERE published = true"

Post.unscoped.all
# Fires "SELECT * FROM posts"

Post.where(published: false).unscoped.all
# Fires "SELECT * FROM posts"
```

all

`all` 方法，本身已经 scope，返回的是 Relation 对象。如果已调用了别的 scope 方法，则没必要使用它，因为默认返回的就已经是所有符合条件的数据。

为什么参数要是 proc 类型？

```
scope :recent, -> { where("created_at > ?", 2.day.ago) }
```

可以看到这里的 `2.day.ago` 是动态生成的，每一次执行的时候才知道结果。不使用 proc 类型的话，这里立即执行，就和想像中的结果不同了。

> Note: 并不是所有的方法都可以做为 scope 的内容，更多内容 Active Record Query Interface

scope 可以调用的时候带参数

```
scope :find_lazy, -> (id) { where(:id => id) }

# 带默认值
scope :recent_applies, ->(day=3){ where("created_at > ?", day.days.ago) }
```

注意，使用后不能再进行链式调用。

Attribute Methods

TODO

Attribute Methods 文件下的内容

提供针对某属性的读写方法。

主要内容

常用实例方法：

```
[]
[]=
```

其它：

```
attributes
attribute_names

attribute_for_inspect

attribute_present?
has_attribute?
```

另外对某个属性 attribute，其它相关模块还会提供以下方法：

```
# 在 Read 模块里定义
attribute

# 在 Write 模块里定义
attribute=
```

```
# 在 Query 模块里定义
attribute?
```

覆盖读写方法

一般情况下，如果要覆盖某个属性的读写方法的话，覆盖的通常是 Read/Write 下的方法，也就是：

```
# 如果要覆盖的话，通常会修改以下方法
user.name
user.name=
```

```
# 以下方法不会覆盖，仍然得到原来的值
user[:name]
user[:name]=
```

像 Enum 提供的 `enum` 和 CarrierWave 提供的 `mount_uploader` 都是这样做的。

区别于 Active Model 下的 Attribute Methods

它与我们平常使用到的读写方法，没有直接联系。

Read

`read_attribute` 根据属性名，获取其值。

和 self[:x] 等价

```
read_attribute(:name)

# 等价于

self[:name]

# 等价于

self.name
```

数据从 `attributes` 里获取，不同于直接 `self.name` 它获取的是真实数据。

如果字段用于存储图片信息，并且我们有默认图片，则没有图片时：`self.image` 返回的是默认图片信息，而 `self[:image]` 返回 `nil` 这才是真实信息，这和 `self.attributes` 里的 `image` 信息一致。

那为什么不用 `attributes[:name]` 而用 `read_attribute[:name]` ？因为性能，前者要把所有的属性都找出来，然后取 `name` 属性；而后者可以直接获取 `name` 属性。

Write

属性名，加后缀 `=` 进行赋值。

区别于 attr_writer, 这里的写和数据库操作有关。

Before Type Cast - 类型转换

同样的数据，每个数据库存储多少有一点不同。 我们'对象.属性'或'类.查询'得到的数据， 未必就是数据库里存放的数据(至少形式上不一样)。 上述两点，虽然差异不大，但多少还是要经过处理的。

实例方法：

```
read_attribute_before_type_cast

x_before_type_cast

attributes_before_type_cast
```

举例：对数字和时间，特别是 boolean 类型的数据, 数据库可能用 true/false, t/f, 1/0, T/F 来表示，而我们获取的只有一种 true/false.

```
class Task < ActiveRecord::Base
end

task.read_attribute('id')                            # => 1
task.read_attribute_before_type_cast('id')           # => '1'

task.read_attribute('completed_on')                  # => Sun, 2
1 Oct 2012
task.read_attribute_before_type_cast('completed_on') # => "2012-
10-21"
task.completed_on_before_type_cast('completed_on')   # => "2012-
10-21"

task.attributes
# => {"id"=>nil, "title"=>nil, "is_done"=>true, "completed_on"=>
Sun, 21 Oct 2012,
      "created_at"=>nil, "updated_at"=>nil}

task.attributes_before_type_cast
# => {"id"=>nil, "title"=>nil, "is_done"=>true, "completed_on"=>
"2012-10-21",
     "created_at"=>nil, "updated_at"=>nil}
```

默认，结果已经经过类型转换。但有时候我们也不希望类型转换，比如：在控制台里，对于'时间'不转换我反而觉得更易读。

Query - 后缀 '?' 问询

加后缀 '?' 进行 boolean 判断。

你还在用：

```
<% if @user.login.blank? %>
  <%= link_to 'login', new_session_path %>
<% end %>

# 或

<% if @user.login.present? %>
  <%= @user.login %>
<% end %>
```

你 Out 了，直接：

```
<% unless @user.login? %>
  <%= link_to 'login', new_session_path %>
<% end %>

<% if @user.login? %>
  <%= @user.login %>
<% end %>
```

每一个 record 属性都有此方法，它可以让我们少敲几个字符。但，除非属性本身就是 boolean 类型，其它类型的判断结果有时候会和想像的不一样，请慎用。不要为了少敲几个字符，增加犯错的几率。

原理是：判断其值是否为 false?、blank? 或 zero?

```
# status 为 integer 类型的字段，当它为 0 时：
self.status.present? => true
self.status?         => false
```

Serialization

`serialize` 指定某个字段的存储类型(默认是 YAML)。

这个类型是可序列化的，如：Array，JSON、Hash (此时请注意于【Store】的 `store` 方法的区别)

使用举例：

```
class Post < ActiveRecord::Base
  serialize :title, Hash
end

# 新建对象时，title 相当于 Hash 的名字

post = Post.new
post.title
# => {}

post.title.class
# Hash

post.title = { name: 'Your Name.' }
post.title[:name]
# => Your Name.
```

存储时，如果类型不符合，会报错。

> 名字和【ActiveRecord::Serialization】相同，注意它们的区别。

Primary Key

类方法：

```
primary_key
primary_key=
```

`primary_key` 主键(又称主关键字)。

是表中的一个或多个字段，它的值用于惟一地标识表中的某一条记录。默认是 'id' 属性，一般不会更改。

使用举例：

```
class Project < ActiveRecord::Base
  self.primary_key = 'sysid'
end

# 或

class Project < ActiveRecord::Base
  def self.primary_key
    'foo_' + super
  end
end

Project.primary_key # => "foo_id"
```

除上述类方法外，还有类方法：

```
dangerous_attribute_method?

define_method_attribute

quoted_primary_key
```

和实例方法：

```
id
id=
id?
id_before_type_cast
id_was

to_key
```

其它

~~Dirty~~

跟踪对象值的变化情况。

Active Model 有同名 Dirty 模块，这里是对它的使用，并且这里没有对外提供 API.

文档可以参考【Model 的增强模块 Dirty 】。

~~Time Zone Conversion~~

TODO

Persistence

很重要的模块，提供保存、更新、删除等操作。

```
# 保存操作
save
save!

# 更新操作
update & update_attributes
update! & update_attributes!
update_attribute
update_column
update_columns

# 删除操作
delete
destroy
destroy!

# 状态询问
new_record?
persisted?
destroyed?

# 更新 updated_at 或指定字段
touch

# 重新加载，清除缓存
reload

# 减一、加一
decrement
decrement!
increment
increment!

# 反转某 bool 属性的值
```

```
toggle
toggle!

# 单表继承时，子类对象行为表现像父类对象
becomes
becomes!
```

> Note: 这里大部分是对单个对象的操作。

数据更新方法对比

方法	使用默认 Accessor?	持久化对象?	校验	回调	更新 updated_at	Readonly 检查
x=	是	否	-	-	-	-
write_attribute	否	否	-	-	-	-
update_attribute	是	是	否	是	是	是
assign_attributes & attributes=	是	否	-	-	-	-
update & update_attributes	是	是	是	是	是	是
update_column	否	是	否	否	否	是
update_columns	否	是	否	否	否	是
User::update	是	是	是	是	是	是
User::update_all	否	是	否	否	否	否

x= 表示直接赋值，其它几个是方法名

write_attribute(:name, ?) 等价于 user[:name]= ?

User::update 是类方法，直接封装了 User#update 实例方法，效果是一样的。

update & update_attributes 封装了 assign_attributes

update_column 直接封装了 update_columns

参考

Different Ways to Set Attributes in ActiveRecord

对比，然后使用合适的方法

- delete vs destroy

前者不会触发回调，后者会。前者速度更快。

- delete_all vs destroy_all

前者会触发回调，后者不会。前者速度更快。

- update_column vs update_attribute

前者不会触发校验，后者会。前者速度更快。

- update_column/update_attribute vs save

前者更新部分，后者更新所有(并且要运行校验、回调)。前者速度更快。

- update_all/find_in_batches vs for/each

前者为批量操作，后者不是。前者速度更快。

- select vs 默认查询

前者指定查询部分字段，后者查询所有字段。前者速度更快。

- pluck and plucks vs map

前者直接指定查询部分字段，后者查询所有字段然后才取部分字段。前者速度更快。

- index vs 默认查询

前者借助了索引，后者没有。前者速度更快。

- 原生 SQL vs AR 方法

使用原生 SQL 比使用 ActiveRecord 方法要快。(不解释)

参考

Rails-with-massive-data

多个 save 方法

在以下几个类或模块里都有 save 方法，那么它到底是如何工作，如何保存数据的呢。

```
module ActiveRecord
  class Base
    # ... ...
    # 眞正的保存 create_or_udpate
    include Persistence

    # ... ...
    # 相关校验 perform_validations
    include Validations

    # ... ...
    # 相关脏数据 Dirty
    include AttributeMethods

    # ... ...
    # 加上事务 rollback_active_record_state! 及 with_transaction_r
eturning_status
    include Transactions

    # ... ...
  end
end
```

根据 Ruby 的代码执行规则：会"反着"模块的引入顺序，嵌套执行里面的 save 方法。

```
进入 Transactions 的 save
进入 AttributeMethods 的 save
进入 Validations 的 save
进入 Persistence 的 save

离开 Persistence 的 save
离开 Validations 的 save
离开 AttributeMethods 的 save
离开 Transactions 的 save
```

同理，当执行其它某个操作的时候，也会发生类似情形。

参考

Life of save in ActiveRecord

Counter Cache

按要求加减指定计数器的值、统计数目的加一、统计数目的减一、重置计数器的值。

```
# 更新计数器的值
update_counters(id, counters)

# 下面这两个方法基于 update_counters
increment_counter(counter_name, id)
decrement_counter(counter_name, id)

# 当计数出错时，可用它来校正
reset_counters(id, *counters)
```

这几条命令直接转化成 sql 语句，所以性能上要比普通的"给对象的计数器赋值，然后保存对象"要快，并且准确性得到了更高的保证。

之前没有统计数目，新增统计数目，或之前的统计数目存在错误，使用 reset_counters 你可以又快、又准确的得到统计数目。

update_counters(id, counters)

计数器实现。在它基础上通常有加一、减一操作，但也可以单独使用。 考虑到并发，这里并不只是Rails层面的get，然后set。而是在SQL层面"真正执行时"才增量加减(但没有用锁机制)。下面的加一、减一方法都是这样。

```
# 注意：参数是字段名
Post.update_counters 3, comments_count: +1

UPDATE "posts" SET "comments_count" = COALESCE("comments_count",
0) + 1 \n
WHERE "posts"."id" = 3

# 原来没有使用计数器，或因为某种原因计数错误。现在我们要使用或修正计数器。
Post.update_counters 3, comments_count: post.comments.count
```

increment_counter(counter_name, id)

给 counter_name 字段进行加一。

decrement_counter(counter_name, id)

给 counter_name 字段进行减一。

reset_counters(id, *counters)

上文提到的"新增计数器"，也可用此方法实现。

```
# 注意：参数是关系表名(可以是多个)，而不是字段名
Post.reset_counters(3, :comments)

SELECT  "posts".* FROM "posts"  WHERE "posts"."id" = ? LIMIT 1
[["id", 3]]
SELECT COUNT(*) FROM "comments"  WHERE "comments"."post_id" = ?
 [["post_id", 3]]
UPDATE "posts" SET "comments_count" = 2 WHERE "posts"."id" = 3
```

基于SQL层面，重置(理解为校正，而不是归零)一个或多个计数器的值。计数器有时候会不准，特别是我们用来计数关联对象的个数，而自己又手动删除它们。

> Note: counter_cache 的 attribute 默认是 read_only

Querying

```
delegate :find, :take, :take!, :first, :first!, :last, :last!, :
exists?, :any?,
         :many?, to: :all
delegate :second, :second!, :third, :third!, :fourth, :fourth!,
:fifth, :fifth!,
         :forty_two, :forty_two!, to: :all
delegate :first_or_create, :first_or_create!, :first_or_initiali
ze, to: :all
delegate :find_or_create_by, :find_or_create_by!, :find_or_initi
alize_by, to: :all
delegate :find_by, :find_by!, to: :all
delegate :destroy, :destroy_all, :delete, :delete_all, :update,
:update_all, to: :all
delegate :find_each, :find_in_batches, to: :all
delegate :select, :group, :order, :except, :reorder, :limit, :of
fset, :joins, :where,
         :rewhere, :preload, :eager_load, :includes, :from, :lock
, :readonly, :having,
         :create_with, :uniq, :distinct, :references, :none, :un
scope, to: :all
delegate :count, :average, :minimum, :maximum, :sum, :calculate,
to: :all
delegate :pluck, :ids, to: :all
```

除上述 delegate 方法外，还有：

```
find_by_sql

count_by_sql
```

其它

~~Null Relation~~

结合 `ActiveRecord::Relation#none` 可以用来表示和处理结果为空的 Relation.

Active Record 工具

为了完成某项任务而生。包括但不限于：

Transactions 事务

- Validations 校验

包括 Associated Validator、Presence Validator、Uniqueness Validator.

Store

No Touching

Readonly Attributes

Nested Attributes 嵌套属性

Integration

Inheritance 单表继承

Enum 枚举

Callbacks 回调

Attributes

- Locking

包括 Optimistic、Pessimistic

~~Translation~~

有多个模块是从 Base 抽取而来，它们是：Attribute Assignment、Dynamic Matchers、Inheritance、Integration、Model Schema、Querying、Readonly Attributes、Sanitization、Scoping、Translation

Callbacks 回调

基于 Action Model 提供的 `define_model_callbacks` 和 Active Support 提供的 `define_callbacks` 方法，共生成十几个过滤器方法。

```
# ActiveRecord::Callbacks
define_model_callbacks :initialize, :find, :touch, :only => :aft
er
define_model_callbacks :save, :create, :update, :destroy

# ActiveModel::Callbacks
define_callbacks :validation

# ActiveRecord::Transactions
define_callbacks :commit, :rollback
```

和 AbstractController::Callbacks::ClassMethods 用元编程生成过滤器的方法名，是两种手法(尽管最终都是基于 ActiveSupport::Callbacks).

是什么？

通过钩子的方式，影响对象的生命周期。

有哪些？

共 22 个：

```
CALLBACKS = [
  :after_initialize, :after_find, :after_touch,
  :before_save, :around_save, :after_save,
  :before_create, :around_create, :after_create,
  :before_update, :around_update, :after_update,
  :before_destroy, :around_destroy, :after_destroy,
  :before_validation, :after_validation,
  :after_commit, :after_rollback,

  :after_create_commit, :after_update_commit, :after_destroy_com
mit
]
```

怎么使用？

调用方式主要有以下几种:

1. 宏定义的方式，后面跟方法名进行调用
2. 传递一个可回调对象
3. 以类方法的形式，传递一个 block

1、3 用得最多，第 2 次之，还有一种方法不推荐(以字符串的形式传递)，第 2 可以起到分离和复用的作用，但复杂度提高了，并且有其它实现手法可替代。

```
# 1 宏定义的方式，后面跟方法名进行调用
class Topic < ActiveRecord::Base
  before_destroy :delete_parents

  private
    def delete_parents
      self.class.delete_all "parent_id = #{id}"
    end
end
```

回调是针对单个 record 对象而言的。当传递给回调的参数是一个实例对象时，把当前 record 对象当做参数，传递并执行实例对象里和回调同名的方法。创建实例对象的时候，你也可以传递参数。

```
# 2 传递一个可回调对象
class BankAccount < ActiveRecord::Base
  before_save      EncryptionWrapper.new
  after_save       EncryptionWrapper.new
  after_initialize EncryptionWrapper.new
end

class EncryptionWrapper
  def before_save(record)
    record.credit_card_number = encrypt(record.credit_card_numbe
r)
  end

  def after_save(record)
    record.credit_card_number = decrypt(record.credit_card_numbe
r)
  end

  alias_method :after_initialize, :after_save

  private
    def encrypt(value)
      # Secrecy is committed
    end

    def decrypt(value)
      # Secrecy is unveiled
    end
end

# 2 传递一个可回调对象
class BankAccount < ActiveRecord::Base
  before_save      EncryptionWrapper.new("credit_card_number")
  after_save       EncryptionWrapper.new("credit_card_number")
  after_initialize EncryptionWrapper.new("credit_card_number")
end

class EncryptionWrapper
  def initialize(attribute)
```

```
    @attribute = attribute
  end

  def before_save(record)
    record.send("#{@attribute}=", encrypt(record.send("#{@attrib
ute}")))
  end

  def after_save(record)
    record.send("#{@attribute}=", decrypt(record.send("#{@attrib
ute}")))
  end

  alias_method :after_initialize, :after_save

  private
    def encrypt(value)
      # Secrecy is committed
    end

    def decrypt(value)
      # Secrecy is unveiled
    end
end
```

```
# 3 以类方法的形式，传递一个 block
class Napoleon < ActiveRecord::Base
  before_destroy { logger.info "Josephine..." }
  before_destroy do
    # some code
  end
  # ...
end
```

抽取封装回调方法

覆盖方法名，重新定义方法内容(注意：这里定义的是实例方法啊，内容不会被执行，其它类再继承才能执行!)

```
# 1
class Topic < ActiveRecord::Base
  def before_destroy() destroy_author end
end

class Reply < Topic
  def before_destroy() destroy_readers end
end

# 2
class PictureFileCallbacks
  def after_destroy(picture_file)
    if File.exist?(picture_file.filepath)
      File.delete(picture_file.filepath)
    end
  end
end

class PictureFile < ActiveRecord::Base
  after_destroy PictureFileCallbacks.new
end

# 3
class PictureFileCallbacks
  def self.after_destroy(picture_file)
    if File.exist?(picture_file.filepath)
      File.delete(picture_file.filepath)
    end
  end
end

class PictureFile < ActiveRecord::Base
  after_destroy PictureFileCallbacks
end
```

怎么取消后面的回调？

在方法里返回 `false`

如何理解 around

用 around_save 举例：

```
def around_save
   # 类似 before save ...
   yield # 执行 save
   # 类似 after save ...
end
```

一个生命周期为一个事务(BEGIN...COMMIT)

如果没有特殊操作，一个生命周期里，对数据库的操作是在同一个“事务”里进行的。所以，如果中间的失败，返回 `false` 那么后续操作，甚至前面的操作都会失败。

开发里，可以在控制台或日志里查看 BEGIN...COMMIT 之间的增删查改语句，确保同一个事务里您的操作是预期的。

回调及其顺序

每个操作，它所对应的回调(按顺序来的)。

创建

```
before_validation
after_validation

before_save
around_save

before_create
around_create
after_create

after_save

after_commit/after_rollback
```

更新

```
before_validation
after_validation

before_save
around_save

before_update
around_update
after_update

after_save

after_commit/after_rollback
```

删除

```
before_destroy
around_destroy
after_destroy

after_commit/after_rollback
```

save = create + update

commit = create + update + destroy 自然地也包含了 save 在内。

> Note: 执行 create 和 update 操作，都会触发 after_save 回调。但它的顺序始终在 after_create 和 after_update 之后。即使在 model 里它定义在前面，效果一样。

after_initialize 和 after_find

不管是直接 new 或其它途径，只要有对象被初始化，就会触发 after_initialize 回调。使用它，可以避免重写 initialize 方法。

只要从数据库里查找记录，就会触发 after_find 回调。并且，after_find 和 after_initialize 同时定义的时候，after_find 优先级要高于 after_initialize.

initalize 和 find 只有 after* 回调，也就是 *after_initialize* 和 *after_find callbacks*，没有对应的 *before** 回调。但它的用法和其它的回调一 样：

```
class User < ActiveRecord::Base
  after_initialize do |user|
    puts "You have initialized an object!"
  end

  after_find do |user|
    puts "You have found an object!"
  end
end

>> User.new
You have initialized an object!
=> #<User id: nil>

>> User.first
You have found an object!
You have initialized an object!
=> #<User id: 1>
```

> Note: 上面例子已经证明，从数据库里查找记录，也会有新的对象创建，所以会有 initialize 过程。

after_touch

对对象执行 touch 更新后，都会运行 after_touch 回调方法。我们可以指定其内容：

```
class User < ActiveRecord::Base
  after_touch do |user|
    puts "You have touched an object"
  end
end

>> u = User.create(name: 'Kuldeep')
=> #<User id: 1, name: "Kuldeep",
          created_at: "2013-11-25 12:17:49", updated_at: "2013-11-25 12:17:49">

>> u.touch
You have touched an object
=> true
```

在 belongs_to 关联里，除 `touch: true` 外，使用 after_touch 可以做更多的操作。

```
class Employee < ActiveRecord::Base
  belongs_to :company, touch: true
  after_touch do
    puts 'An Employee was touched'
  end
end

class Company < ActiveRecord::Base
  has_many :employees
  after_touch :log_when_employees_or_company_touched

  private
  def log_when_employees_or_company_touched
    puts 'Employee/Company was touched'
  end
end

>> @employee = Employee.last
=> #<Employee id: 1, company_id: 1,
              created_at: "2013-11-25 17:04:22", updated_at: "20
13-11-25 17:05:05">

# triggers @employee.company.touch
>> @employee.touch
Employee/Company was touched
An Employee was touched
=> true
```

> Note: after_touch 实际上运用得比较少。执行 touch 操作，除它之外，还会触发 after_commit 和 after_rollback 回调函数。

查看某个记录关联的回调及其顺序

以 save 举例：

```
after_save :回调 1, :回调 2
```

查看某个记录关联的回调及其顺序

```
a_record._save_callbacks.map{ |c| puts c.raw_filter };

=>
  save 回调 1
  save 回调 2
  ... ...
  save 回调 3
  save 回调 4
```

执行顺序从下往上，所以回调之间有相同内容的话，上面的可以覆盖下面的。但如果下面的回调执行失败的话，也会影响到上面的。

这里不区分是系统生成的方法(大多数是关联时就带有，如 autosave_associated_records、belongs_to_counter_cache、has_many_dependent)，还是我们自定义的方法。

用 `:prepend` 参数可以更改回调在堆栈里的顺序，如：

```
class Foo < ActiveRecord::Base
  belongs_to :bar, autosave: true

  before_save :modify_bar, prepend: true
end
```

在这里，autosave 在 modify_bar 之前完成，也就是说 modify_bar 优先级高。

跳过回调

主要有 4 种方式：

update_columns 和 update_column
它们都是直接执行 SQL 语句，不会触发回调方法。

使用举例：

```
user.update_columns(last_request_at: Time.current)
```

save(:validate => false)

跳过 Model 里的所有校验。

skip_callback
跳过某个回调。

使用举例：

```
class Writer < Person
  skip_callback :validate, :before, :check_membership, if: -> {
self.age > 18 }
end
```

x_without_callbacks
以 object.send(:x_without_callbacks) 跳过某个系列的回调。

使用举例：

```
object.send(:create_without_callbacks)
object.send(:update_without_callbacks)
```

可选参数

:on

使用 :on 指定要关联的事件。

```
class User < ActiveRecord::Base
  before_validation :normalize_name, on: :create

  # :on takes an array as well
  after_validation :set_location, on: [ :create, :update ]

  protected
    def normalize_name
      self.name = self.name.downcase.titleize
    end

    def set_location
      self.location = LocationService.query(self)
    end
end
```

:prepend

```
class Topic < ActiveRecord::Base
  has_many :children, dependent: destroy

  before_destroy :log_children

  private
    def log_children
      # Child processing
    end
end
```

topic 已经被删除了，没办法 log_children，可以改为这样：

```
class Topic < ActiveRecord::Base
  has_many :children, dependent: destroy

  before_destroy :log_children, prepend: true

  private
    def log_children
      # Child processing
    end
end
```

:if 和 :unless

```
class Order < ActiveRecord::Base
  before_save :normalize_card_number, if: :paid_with_card?
end

class Order < ActiveRecord::Base
  before_save :normalize_card_number, if: "paid_with_card?"
end

class Order < ActiveRecord::Base
  before_save :normalize_card_number,
    if: Proc.new { |order| order.paid_with_card? }
end

class Comment < ActiveRecord::Base
  after_create :send_email_to_author, if: :author_wants_emails?,
    unless: Proc.new { |comment| comment.article.ignore_comments
? }
end
```

after_create

after_create 从名字上看，record 对象应该已经创建并保存到数据库了。但实际情况却并非总是如此，因为 after_create 和整个创建、保存是处于同一个生命周期的（共用一个 commit），所以还是有可能保存失败。

坑在哪？如果我们在此回调需要调用了相关 record 对象，并且认为它已经保存到数据库了。例如用户创建后，即刻发送邮件给Ta，即使是异步的，也有可能会报找不到此对象。因为有一点点时间差，record 对象还未眞正保存到数据库，但已经被当做保存数据库进而调用了。

after_commit on: :create

使用它，则保证了 record 对象已经眞正被保存了，保存到数据库。似乎解决了上述问题。

坑在哪？commit 里不能写和 commit 自己的代码。比如你要 save 一个 record 对象，那么不能再在 after_commit on: :create 执行和 save 有关的操作，如果违反了，就会陷入死循环里。可以使用 update_columns 等不会触发 commit 的方法更新自己，或者你 save 其它对象。

Nested Attributes 嵌套属性

提供类方法 `accepts_nested_attributes_for(*attr_names)`

attr_names 由：一个或多个属性(association_name) 和 一个或多个可选参数(option)组成。

只接受 options：

```
:allow_destroy
:reject_if
:limit
:update_only
```

`:limit` 在单个 record 构建时不执行；指量构建时，在校验之前执行，并且超过限制的话直接抛错误。实际项目中，不推荐使用。

当你声明嵌套属性时，**Rails** 会自动帮你定义属性的写方法。

```
# 摘录部分代码
def #{association_name}_attributes=(attributes)
  assign_nested_attributes_for_#{type}_association(:#{associatio
n_name}, attributes)
end
```

`association_name` 就是你声明的属性，例如：

```
class Book < ActiveRecord::Base
  has_one :author
  has_many :pages

  accepts_nested_attributes_for :author, :pages
end
```

生成 `author_attributes=(attributes)` 和 `pages_attributes=(attributes)`

对于关联对象，会自动设置 `:autosave`

```
# 摘录部分代码
reflection.autosave = true                          # 自动保存
add_autosave_association_callbacks(reflection) # 回调在自动保存时仍然有效
```

对于嵌套的属性，默认你可以执行写操作，但不能删除它们。

如果你真的要这么做，也可以通过 `:allow_destroy` 来设置。如：

```
class Member < ActiveRecord::Base
  has_one :avatar
  accepts_nested_attributes_for :avatar, allow_destroy: true
end

# 然后

member.avatar_attributes = { id: '2', _destroy: '1' }
member.avatar.marked_for_destruction? # => true
member.save
member.reload.avatar # => nil
```

上面举例是一对一，下面的一对多关系类似：

```
params = { member: {
  name: 'joe', posts_attributes: [
    { title: 'Kari, the awesome Ruby documentation browser!' },
    { title: 'The egalitarian assumption of the modern citizen'
},
    { title: '', _destroy: '1' } # this will be ignored
  ]
}}

member = Member.create(params[:member])
member.posts.length # => 2
member.posts.first.title # => 'Kari, the awesome Ruby documentat
ion browser!'
member.posts.second.title # => 'The egalitarian assumption of th
e modern citizen'
```

自动保存多个嵌套属性，有的可能不符合校验。

为了处理这种情况。你可以设置 `:reject_if` :

```
class Member < ActiveRecord::Base
  has_many :posts

  accepts_nested_attributes_for :posts, reject_if: proc do |attr
ibutes|
    attributes['title'].blank?
  end
end

params = { member: {
  name: 'joe', posts_attributes: [
    { title: 'Kari, the awesome Ruby documentation browser!' },
    { title: 'The egalitarian assumption of the modern citizen'
},
    { title: '' } # this will be ignored because of the :reject_
if proc
  ]
}}

member = Member.create(params[:member])
member.posts.length # => 2
member.posts.first.title # => 'Kari, the awesome Ruby documentat
ion browser!'
member.posts.second.title # => 'The egalitarian assumption of th
e modern citizen'
```

在这里，效果和上面使用 `_destroy: '1'` 有类似之处。

重现 **autosave** 创建过程

```
Book has_many :pages

reflection = Book._reflect_on_association(:pages)
Book.send(:add_autosave_association_callbacks, reflection)

Book.reflect_on_all_autosave_associations
```

update_only - 解决更新关联对象时的困扰

之前的写法：

```
# alias.rb
class Alias < ActiveRecord::Base
  belongs_to :user
  accepts_nested_attributes_for :user
end
```

举例：更新关联对象

```
# 传递 id
Alias.first.user.name
>> "Alice"
Alias.first.update_attributes(
{
  :user_attributes => {
    :id => 1, # <- 这行
    :name => "Bob"
  }
})

Alias.first.user.name
>> "Bob"

# 外键用的是传递过来的 id
Alias.first.user_id
>> 1

# 不传递 id
Alias.first.update_attributes(
{
  :user_attributes => {
    :name => "Bob"
  }
})

# 没有传递外键，则会先删除，然后重新创建关联对象
Alias.first.user_id
>> 2
```

上述情况，虽然文档上已经写明了。但这违背了我们的直觉，建议加上 `:update_only` 参数。

之后的写法：

```
# alias.rb
class Alias < ActiveRecord::Base
  belongs_to :user
  accepts_nested_attributes_for :user, update_only: true
end
```

update_only 仅作用于单一关系，对 collection 使用无效。

当然，除上述方法外，还有解决办法就是：直接获取，然后操作被关联的对象作。或者，直接通过关联对象进行赋值，然后保存（利用 auto_save 进行自动更新）。

---

invert_of 的另一个作用 accepts_nested_attributes_for with Has-Many-Through Relations

allow_destroy 选项的使用 【Rails】fields_for と accepts_nested_attributes_for 和此方法配套使用的是 `fields_for` 方法。

Inheritance 单表继承

单表继承 一个或多个 Model 继承于另一个 Model，并且最终它们用的是同一张表。

我们会遇到这样的情况：

Employee 有 manager 和 developer
Computer 有 pc 和 mac

有时候，需要把它区分对待；有时候，又要对它们一视同仁。

设计表和 model 时，用得比较多的是"两张类似的表，两个类似的 model" 还是 "一张表，一个 model，一个用于标识的字段"，再或者是第 3 种选择，也就是这里要讲的"STI(单表继承)"。

一般说来，单表继承通常用于：属性一样，但行为不一致。

是什么？

单表，就是在数据库你只需要一张表。
继承，就是你的 model 之间存在着继承关系。

上面的例子中：
我们可以只用 employees 表，却有 model Manager 和 model Developer，它们都继承于 Employee.
我们可以只用 computers 表，却有 model Pc 和 model Mac，它们都继承于 Computer.

选择

1. 各子模块属性是一样的。这里要从"面向对象"的角度去看，而不是简单的'属性一样'就能使用。

2. 需要把这些子模块代表的数据放在一想显示或者说做聚合吗？如果需要的话，那么使用 STI 是比较好的选择。因为从性能上来说，即使优化做得再好，跨表操作，也不如在一个表里操作，来得简单、高效。

3. 和第 1 条有点类似，它们属性是一样的，只是行为不一致而矣。如果只是大部分属性一样，那么可以考虑一下"多态关联"。

实际使用过程中，单表继承在某些方面提供了方便(比如：自动设置 type)，但同时也会造成麻烦(比如：多个 model)。同样的，使用"两张类似的表，两个类似的 model" 或 "一张表，一个 model，一个用于标识的字段"或其它手段，也会有自己的问题。

这些方法都有利有弊，选择时，我们尽量选择利大于弊的那一种吧。

使用

单表继承默认使用 `type` 做为标识字段，在表里面新增字段即可，当然，也可以用【Model Schema】里的 `inheritance_column` 自定义标识字段。创建相应对象时，会根据所使用的模块名自动设置它的值。

```
class Company < ActiveRecord::Base
  # ...
end

class Firm < Company
  # ...
end

class Client < Company
  # ...
end
```

Transactions 事务

`transaction(options = {}, &block)` 要么同时成功，往下走；要么同时失败，回滚。

事务是恢复和并发控制的基本单位。

事务应该具有4个属性：原子性、一致性、隔离性、持久性。这四个属性通常称为 ACID 特性。

- 原子性(atomicity). 一个事务是一个不可分割的工作单位，事务中包括的诸操作要么都做，要么都不做。
- 一致性(consistency). 事务必须是使数据库从一个一致性状态变到另一个一致性状态。一致性与原子性是密切相关的。
- 隔离性(isolation). 一个事务的执行不能被其他事务干扰。即一个事务内部的操作及使用的数据对并发的其他事务是隔离的，并发执行的各个事务之间不能互相干扰。
- 持久性(durability). 持久性也称永久性(permanence)，指一个事务一旦提交，它对数据库中数据的改变就应该是永久性的。接下来的其他操作或故障不应该对其有任何影响。

使用举例：

```
ActiveRecord::Base.transaction do
  david.withdrawal(100)
  mary.deposit(100)
end
```

Rails 提供的事务，可以做为类方法，也可以做为实例方法进行调用。
并且，一个事务里面可以操作多个对象：

```
# 类方法
Account.transaction do
  balance.save!
  account.save!
end
```

```
# 实例方法
balance.transaction do
  balance.save!
  account.save!
end
```

`after_commit(*args, &block)` 和 `after_rollback(*args, &block)` 使用上完全一样。

默认 commit 包括： created, updated, 和 destroyed. 不过，你可以用可选参数 `:on` 指定其中的一个或多个:

```
after_commit :do_foo, on: :create
after_commit :do_bar, on: :update
after_commit :do_baz, on: :destroy

after_commit :do_foo_bar, on: [:create, :update]
after_commit :do_bar_baz, on: [:update, :destroy]
```

原理，和普通的回调类似。使用 Active Support 提供的 `set_callback(name, *filter_list, &block)` 完成。

使用事务后，其它 destroy、save、save!、touch 等操作时也会对应的受到影响。

Locking

锁，分为悲观锁(Pessimistic)和乐观锁(Optimistic).

悲观锁(Pessimistic)

特性

强烈的独占和排他。

它指的是对数据被外界(包括本系统当前的其他事务，以及来自外部系统的事务处理)修改持保守态度，因此，在整个数据处理过程中，将数据处于锁定状态。

实现

往往依靠数据库提供的锁机制(也只有数据库层提供的锁机制才能真正保证数据访问的排他性，否则，即使在本系统中实现了加锁机制，也无法保证外部系统不会修改数据)。

一个典型的依赖数据库的悲观锁调用：

```
select * from account where name="Erica" for update;
```

这条 sql 语句锁定了 account 表中所有符合检索条件(name="Erica")的记录。 本次事务提交之前(事务提交时会释放事务过程中的锁)，外界无法修改这些记录。

> 注意：根据 name 是否有索引、是否是唯一索引、是否是主键，决定锁全表、区间、单个记录。

Rails 的悲观锁

相关方法有： `lock` 、 `lock!` 和 `with_lock` .

其中，lock 和 with_lock 都是封装 lock! 而来。

lock 相当于 lock! 的别名，但调用者可以是 relation 对象。
with_lock 和事务捆绑在一起，并且参数可以是代码块。

使用举例：

```
# 使用 lock，注意生成的 SQL
Account.lock.find(1)
# SELECT `accounts`.* FROM `accounts` WHERE `accounts`.`id` = 1
LIMIT 1 FOR UPDATE

# lock 结合 transaction 一起使用
Account.transaction do
  # select * from accounts where name = 'shugo' limit 1 for upda
te
  shugo = Account.where("name = 'shugo'").lock(true).first
  yuko = Account.where("name = 'yuko'").lock(true).first
  shugo.balance -= 100
  shugo.save!
  yuko.balance += 100
  yuko.save!
end

# 使用 lock!
Account.transaction do
  # select * from accounts where ...
  accounts = Account.where(...)
  account1 = accounts.detect { |account| ... }
  account2 = accounts.detect { |account| ... }

  # account1 和 account2 只能是单个对象
  # select * from accounts where id=? for update
  account1.lock!
  account2.lock!
  account1.balance -= 100
  account1.save!
  account2.balance += 100
  account2.save!
end

# 使用 with_lock!
account = Account.first

# account 加上了锁，代码块加上了事务
```

```
account.with_lock do
  account.balance -= 100
  account.save!
end
```

使用注意

1) 锁表，一个地方执行写的时候，另一个地方不能同时执行写操作，这没问题。但问题是，你也不能执行读操作。

2) 一个地方读数据，并赋值给对象。另一个地方在这之后执行了写操作，这个对象(脏数据)会覆盖已经更新过的数据，而不是报错。

> Note: 它是数据库级别的锁。

乐观锁(Optimistic)

特性

乐观锁(Optimistic Locking) 相对悲观锁而言，乐观锁机制采取了更加宽松的加锁机制。

悲观锁大多数情况下依靠数据库的锁机制实现，以保证操作最大程度的独占性。但随之而来的就是数据库性能的大量开销，特别是对长事务而言，这样的开销往往无法承受。而乐观锁机制在一定程度上解决了这个问题。

实现

大多是基于数据版本(Version)记录机制实现。

何谓数据版本？即为数据增加一个版本标识，在基于数据库表的版本解决方案中，一般是通过为数据库表增加一个 "version" 字段来实现。读取出数据时，将此版本号一同读出，之后更新时，对此版本号 +1. 此时，将提交数据的版本数据与数据库表对应记录的当前版本信息进行比对，如果提交的数据版本号大于数据库表当前版本号，则予以更新，否则认为是过期数据。

Rails 的乐观锁

使用举例：

1) 给表添加 `:lock_version` 属性。

```
add_column :products, :lock_version, :integer, :default => 0, :n
ull => false
```

2) 在表单里使用此属性。(此处略)

3) 已经生效。如果更新的是脏数据，会报错 `StaleObjectError` ，可根据这个做相应处理。

```
p1 = Product.find(1)
p2 = Product.find(1)

p1.name = "Michael"
p1.save

p2.name = "should fail"
p2.save
# 如果别人想再次更改，(脏数据)不会覆盖已经更新过的数据，而是会报错。
# => Raises a ActiveRecord::StaleObjectError
```

更改约定：

1) 默认标识字段是 `lock_version` ，当包含此属性时，按照约定乐观锁会"自动生效"。有时候(比如：遗留项目已经使用此字段，但却不是用于"锁")，我们可能需要拒绝"自动生效"，可以配置：

```
ActiveRecord::Base.lock_optimistically = false

# 或，只针对某个 model
ClassName.lock_optimistically = false
```

2) 可以用 `locking_column` 更换默认的标识字段，如：

```
class Person < ActiveRecord::Base
  self.locking_column = :lock_person
end
```

> Note: 它是应用级别的锁。

Enum 枚举

元编程生成一系列的方法：

```
definitions.each do |name, values|
  klass.singleton_class.send(:define_method, name.to_s.pluralize
)

  define_method("#{name}=")
  define_method(name)

  define_method("#{name}_before_type_cast")

  pairs.each do |value, i| # pairs 约等于 values
    define_method("#{value}?")
    define_method("#{value}!")
    klass.scope value
```

假设，我们有 `status` 字段，使用 `enum` 后会生成什么方法？

```
class Post < ActiveRecord::Base
  enum status: [ :active, :archived ]
end
```

1 同名类方法(复数形式)

```
Post.statuses
# => {"active"=>0, "archived"=>1}
```

2 与 value 同名的 scope 方法

```
Post.active
# => SELECT "posts".* FROM "posts"  WHERE "posts"."status" = 0
```

3 value? 询问是否为某值

```
post.active?
=> false
```

4 value! 更新为某值

```
post.active!

begin transaction
  UPDATE "posts" SET "status" = ?, "updated_at" = ? \n
  WHERE "posts"."id" = 2  [["status", 0],
                           ["updated_at", "2014-04-20 09:06:53.7
22202"]]
commit transaction
=> true
```

5 同名实例方 法(get 类型)

```
post.status
=> "active"
```

注意：此处这里如果用 `post[:status]` 的方式获取属性值，返回的值和数据库中保存的一样，是数字。

6 同名实例方法= (set 类型)

```
post.status = "archived"
=> "archived"
```

注意：此处一定要区别于 `post[:status]=` 设置属性值。

7 同名实例方法(get 类型)

```
post.status_before_type_cast
=> "archived"
```

注意 enum 的字段在数据库保存的是 integer 类型，但在外表现的却是字符串，我们查询、更新的都以字符串的形式进行。

另外：

注意 enum 生成的类方法、实例方法不要与 Active Record 提供的方法，及同一个 Model 下其它 enum 生成的方法有重复。

每个 Model 下面都会有一个叫 `defined_enums` 的变量用来记录 enum 相关信息。

读取、设置某属性的值，通常有两种方式：

```
user.name
user.name=

# 或

user[:name]
user[:name]=
```

一般情况下，它们的值是一样的。但这里的 Enum 打开了 `user.name` 和 `user.name=` 方法，变换了返回结果，这会导致两者的值不一样。

另外，要注意当使用 `update` 更新多个属性的时候，操作的是数字，而非字符串。同理，使用 `where` 查询多个属性的时候，操作的是数字，而非字符串。

Store

`store(store_attribute, options = {})` 以 JSON(也可以理解为 Hash)的形式存储某字段。

举例，我们数据库里有 `name` 字段，我们想这样存储：

```
name = { last_name: "Kelby", first_name: "Lee" }
```

```
class User < ActiveRecord::Base
  store :name, accessors: [ :last_name, :first_name ], coder: JS
ON
end

u = User.new(last_name: 'Kelby', first_name: 'Lee')
u.last_name                # 直接读/写 key
u.name[:last_name] = 'zk' # 通过 store 的属性来读/写 key

# 通过 store 的属性来读/写时，key 类型可以是 Symbol 或 String
u.settings[:last_name]  # => 'zk'
u.settings['last_name'] # => 'zk'
```

`store` 由【Serialization】的 `serialize` 和下面的 `store_accessor` 两方法组成。

`store_accessor(store_attribute, *keys)` 给已经存在数据的 store 添加 key.

```
class User < ActiveRecord::Base
  store_accessor :name, :nickname
end
```

它主要是增加了和 key 同名的读/写(实例)方法。

`stored_attributes` 查询一个字段有哪些可用的 key.

```
User.stored_attributes[:name] # [:last_name, :first_name, :nickn
ame]
```

> Note: 通过 store 的属性来读/写 key，这里的 key 可以不在 accessors 范围里。如果不在范围里，则不能直接读/写 key，并且 stored_attributes 查看不到。

存数组用上一章节【AttributeMethods Serialization】里的 `serialize` 方法

```
class Comment < ActiveRecord::Base
  serialize :stuff
end

comment = Comment.new  # stuff: nil
comment.stuff = ['some', 'stuff', 'as array']
comment.save
comment.stuff # => ['some', 'stuff', 'as array']
```

Validations 校验

和 Active Model 里的校验实现原理类似，但有一点不同：这里校验的属性要是关联对象或要从数据库'读'数据。

1) validates_associated(*attr_names)

校验是否存在关联(关系)。可以同时校验多个关联(关系)

```
class Book < ActiveRecord::Base
  has_many :pages
  belongs_to :library

  validates_associated :pages, :library
end
```

2) validates_presence_of(*attr_names)

校验(数据库里)是否存在着关联对象。

3) validates_uniqueness_of(*attr_names)

校验属性的值是否唯一。默认是对所有 record 进行校验，可以用 scope 或 conditions 指定约束条件。

```
class Person < ActiveRecord::Base
  validates_uniqueness_of :user_name
end

# 加 scope 约束条件
# 同一 account 的 person，user_name 不能相同
# 不同 account 的 person，user_name 可以相同
class Person < ActiveRecord::Base
  validates_uniqueness_of :user_name, scope: :account_id
end

# 加 conditions 约束条件
# status 为 archived 的 article，title 不能相同
# status 为其它值的 article，title 可以相同
class Article < ActiveRecord::Base
  validates_uniqueness_of :title, conditions: -> { where.not(sta
tus: 'archived') }
end
```

同名实例方法：

```
save
save!

valid? & validate
```

这里的 `save` 是对 Persistence(持久化)里的 save 方法做的一层包装，在"保存"之前用来做校验工作，并不是真正的保存操作。当传递 `validate: false` 时，可以跳过此校验。其它同名实例方法意义类似。

Secure Token

提供 `has_secure_token(attribute = :token)` 类方法

```
class User < ActiveRecord::Base
  has_secure_token
  has_secure_token :auth_token
end
```

使用它可以给某些属性赋值 SecureRandom::base58 长度的字符串。

默认约定使用 `:token` 属性，并生成了对应的 `regenerate_token` 方法：

```
user = User.new
user.save

user.token # => "pX27zsMN2ViQKta1bGfLmVJE"
user.auth_token # => "77TMHrHJFvFDwodq8w7Ev2m7"
```

内部实现：

通过 `before_create` 及属性自带的读写方法完成赋值操作；
不按照约定来，后续也可以更改属性名。可用元编程生成的 `regenerate_#{attribute}` 方法进行更改属性值：

```
user.regenerate_token # => true
user.regenerate_auth_token # => true
```

Integration

实例方法： `to_param`

默认，Rails 生成 URL 时用的是 `primary key` ，也就是数据库里的 `id` 属性。例如：

```
user = User.find_by(name: 'Phusion')
user_path(user)  # => "/users/1"
```

这对于 SEO 和人类识别都不是很友好。我们可以重写 `to_param` 方法，设置更友好的内容：

```
class User < ActiveRecord::Base
  def to_param  # overridden
    name
  end
end

user = User.find_by(name: 'Phusion')
user_path(user)  # => "/users/Phusion"
```

实例方法： `cache_key(*timestamp_names)`

返回一个能够标识对象的字符串：

```
Product.new.cache_key     # => "products/new"
Product.find(5).cache_key # => "products/5" (updated_at not available)
Person.find(5).cache_key  # => "people/5-20071224150000" (updated_at available)
```

当我们需要缓存的地方很多时，默认生成字符串的规则可能满足不了我们的需求。我们可以传递参数给它，用新的规则生成字符串：

```
Person.find(5).cache_key(:updated_at, :last_reviewed_at)
```

类方法： `to_param`

功能上和实例方法 `to_param` 一样，使用举例：

```
class User < ActiveRecord::Base
  to_param :name
end

user = User.find_by(name: 'Fancy Pants')
user.id         # => 123
user_path(user) # => "/users/123-fancy-pants"
```

No Touching

常用方法：

```
no_touching
```

使用举例：

```
ActiveRecord::Base.no_touching do
  Project.first.touch # 不会执行 touch
  Message.first.touch # 不会执行 touch
end

Project.no_touching do
  Project.first.touch # 不会执行 touch
  Message.first.touch # 会对 message 执行 touch; 但它 touch: true 的关联对象不会被 touch
end
```

除上述外，还有方法：

```
no_touching?

touch
# 优先级大于 Persistence 里的 touch 同名方法；
# 如果 no_touching? => true 则不调用 Persistence 里的 touch 方法。
```

Touch Later

提供 `touch_later` 方法。

原来的 `touch` 是一步到位，更新 updated_at 并保存进数据库。
使用 `touch_later` 后，操作分为了两步：1）更新 updated_at 2）保存进数据库。

通常的，它和事务(transaction)一起使用，或者是 belongs_to 并 touch 关联对象。

Attributes

提供方法：

```
attribute

define_attribute
```

下面主要讲解 `attribute` 方法。

1) 覆盖原有类型的行为

使用举例：

```
# db/schema.rb
create_table :store_listings, force: true do |t|
  t.decimal :price_in_cents
end

# app/models/store_listing.rb
class StoreListing < ActiveRecord::Base
end

store_listing = StoreListing.new(price_in_cents: '10.1')

# before
store_listing.price_in_cents # => BigDecimal.new(10.1)

class StoreListing < ActiveRecord::Base
  attribute :price_in_cents, Type::Integer.new
end

# after
store_listing.price_in_cents # => 10
```

2) 重新定义类型

使用举例：

```
class MoneyType < ActiveRecord::Type::Integer
  # 新类型需实现 type_cast 方法
  def type_cast(value)
    if value.include?('$')
      price_in_dollars = value.gsub(/\$/, '').to_f
      price_in_dollars * 100
    else
      value.to_i
    end
  end
end

class StoreListing < ActiveRecord::Base
  attribute :price_in_cents, MoneyType.new
end

store_listing = StoreListing.new(price_in_cents: '$10.00')
store_listing.price_in_cents # => 1000
```

> Note: 覆盖或重新定义新的类型，也许并不是好的实践，使用之后会遇到新的问题。

Readonly Attributes

提供方法：

```
# 声明某些属性为只读
attr_readonly
```

```
# 获取所有只读的属性
readonly_attributes
```

`attr_readonly` 和其它 attr_x 类似，只不过这里设置的是某属性为只读。注意，这里不是校验，所以保存出错的话，不会放到 record 对象的 errors 里。

Suppressor

某个类使用 `suppress` 方法后，会对后续 block 里的保存(save)操作进行处理。

这里影响的是同类型的实例对象，无论是新建、更新操作表面上都执行了，但实际上没有执行。

```
user = User.create! token: 'asdf'

User.suppress do
  user.update token: 'ghjkl'
  assert_equal 'asdf', user.reload.token

  user.update! token: 'zxcvbnm'
  assert_equal 'asdf', user.reload.token

  user.token = 'qwerty'
  user.save
  assert_equal 'asdf', user.reload.token

  user.token = 'uiop'
  user.save!
  assert_equal 'asdf', user.reload.token
end
```

~~Sanitization~~

```
expand_hash_conditions_for_aggregates

sanitize_conditions & sanitize_sql & sanitize_sql_for_conditions

sanitize_sql_array

sanitize_sql_for_assignment

sanitize_sql_for_order

sanitize_sql_hash_for_assignment

sanitize_sql_like
```

Associations 关联

```
Associations 文件下
    4 个关联类方法 -- (1)

Associations 目录下 -- 实现几个关联方法。
    # 7 个 Builder 文件
    builder -- (2)
        HasAndBelongsToMany
        Association
            SingularAssociation
                HasOne
                BelongsTo
            CollectionAssociation
                HasMany
        4 个关联类方法，直接调用它们

    # 11 个 _Association 文件。实现对关联对象的操作！
    Association -- (3)
        SingularAssociation --(3)
            HasOneAssociation
                HasOneThroughAssociation
                    include ThroughAssociation
            BelongsToAssociation
                BelongsToPolymorphicAssociation
        CollectionAssociation --(3)
            HasManyAssociation
                HasManyThroughAssociation
                    include ThroughAssociation
    ThroughAssociation -> HasOneThroughAssociation + HasManyThro
ughAssociation
    ForeignAssociation -> HasOneAssociation + HasManyAssociation

    # 其它

    CollectionProxy(*) -> CollectionAssociation
        继承于 Relation
```

```
    AssociationScope -> CollectionAssociation + SingularAssociat
ion + Association

    JoinDependency(实现 joins) -> FinderMethods + QueryMethods
        JoinPart
            JoinBase
            JoinAssociation

    AliasTracker -> AssociationScope + JoinDependency

    Preloader --(实现 includes, preload, eager_load)
        ThroughAssociation
        Association
            CollectionAssociation
                HasMany
                HasManyThrough
                    include ThroughAssociation
            SingularAssociation
                BelongsTo
                HasOne
                HasOneThrough
                    include ThroughAssociation

Aggregations

AutosaveAssociation

# Reflection 文件
AbstractReflection
    ThroughReflection
        PolymorphicReflection
    MacroReflection
        AggregateReflection
        AssociationReflection
            HasManyReflection
            HasOneReflection
            BelongsToReflection
            HasAndBelongsToManyReflection
```

Associations 文件 - 4 个关联方法

Association 提供我们 4 个常用方法：

```
has_many
has_one
belongs_to
has_and_belongs_to_many
```

其实现步骤

步骤一：

```
调用对应的 Builder，父 Builder, 顶级 Builder …（交叉进行的）
```

步骤二：

```
# 偏向于关联两者
调用对应的 Reflection，父 Reflection, 再父 Reflection，顶级 Reflecti
on …（交叉进行的）

# 偏向于提供方法
通过 association（由最初的 Association 提供）调用对应的 Association
提供的方法 …（交叉进行的）
```

关于 **association** 方法

另，对内部实现来说非常重要的实例方法：

```
association(name)
```

通过它调用对应 Association 提供的方法。
通过它调用 CollectionProxy 提供的方法。

Aggregations - composed_of 方法

我们在一张表里有几个类似字段，比如 customers 表有 address_street, address_city 字段，用于保存地址信息。好的做法，当然是把它们拆分出来，单独做成 address 表。但如果我们不想/能拆分表成的话(改动太大，处理遗留问题等)，使用 `composed_of` 可以实现不用真正拆分表，又能起到到分离的作用。

```
class Customer < ActiveRecord::Base
  composed_of :address, mapping: [ %w(address_street street), %w
(address_city city) ]
end
```

这里，把 address 当做关联对象。原 customer 的 address_street 和 address_city 分别映射成为 address 的 street 和 city 属性。

根据"约定优于配置"，关联对象 address 对应 class Address，我们实现它：

```
class Address
  attr_reader :street, :city

  def initialize(street, city)
    @street, @city = street, city
  end
end
```

之后即可对 Address 的实例对象进行操作。

如何使用？

Customer 有 balance，address_street、address_city 字段。

```
class Customer < ActiveRecord::Base
  # 把 balance 当做关联对象，amount 映射成为它的属性；对应着 class Money

  composed_of :balance, class_name: "Money", mapping: %w(balance amount)

  # 把 address 当做关联对象，street 和 city 映射成为它的属性；对应着 class Address
  composed_of :address, mapping: [ %w(address_street street), %w(address_city city) ]
end
```

可选参数 `:class_name, :mapping, :allow_nil, :constructor, :converter` ，此外，你有下列读、写方法：

```
# reader_method(name, class_name, mapping, allow_nil, constructor)
# writer_method(name, class_name, mapping, allow_nil, converter)

Customer#balance, Customer#balance=(money)
Customer#address, Customer#address=(address)
```

除了读、写方法外，composed_of 还创建管理了 Reflection 关联两者：

```
reflection = ActiveRecord::Reflection.create(:composed_of, part_id, nil, options, self)
Reflection.add_aggregate_reflection self, part_id, reflection
```

注意：我们没有 model Money 和 model Address，也没有它们对应的表，所以要实现其对应的 class，类似：

```
class Money
  include Comparable
  attr_reader :amount, :currency
  EXCHANGE_RATES = { "USD_TO_DKK" => 6 }
```

```
  def initialize(amount, currency = "USD")
    @amount, @currency = amount, currency
  end

  def exchange_to(other_currency)
    exchanged_amount = (amount *
                        EXCHANGE_RATES["#{currency}_TO_#{other_c
urrency}"]).floor
    Money.new(exchanged_amount, other_currency)
  end

  def ==(other_money)
    amount == other_money.amount && currency == other_money.curr
ency
  end

  def <=>(other_money)
    if currency == other_money.currency
      amount <=> other_money.amount
    else
      amount <=> other_money.exchange_to(currency).amount
    end
  end
end

class Address
  attr_reader :street, :city

  def initialize(street, city)
    @street, @city = street, city
  end

  def close_to?(other_address)
    city == other_address.city
  end

  def ==(other_address)
    city == other_address.city && street == other_address.street
  end
end
```

```
# 关键点
class ClassName
  attr_reader :attr1, :attr2

  def initialize(attr1, attr2)
    @attr1, @attr2 = attr1, attr2
  end
end
```

然后就能这么操作：

```
customer = Customer.new

customer.balance
=> #<Money:0x007f8dabd8c940 @amount=nil, @currency="USD">

# 实例化 customer 的 balance 关联对象
customer.balance = Money.new(20)     # sets the Money value objec
t and the attribute
customer.balance                     # => Money value object
customer.balance.amount              # => 20
customer.balance.currency            # => "USD"
customer.balance.exchange_to("DKK") # => Money.new(120, "DKK")
customer.balance > Money.new(10)     # => true
customer.balance == Money.new(20)    # => true
customer.balance < Money.new(5)      # => false

# 还有
customer.address_street = "Hyancintvej"
customer.address_city   = "Copenhagen"
# 实例化 customer 的 address 关联对象
customer.address        # => Address.new("Hyancintvej", "Copenha
gen")
customer.address.street # => "Hyancintvej"
customer.address.city   # => "Copenhagen"

customer.address_street = "Vesterbrogade"
customer.address        # => Address.new("Hyancintvej", "Copenha
gen")
customer.clear_aggregation_cache
customer.address        # => Address.new("Vesterbrogade", "Copen
hagen")

customer.address = Address.new("May Street", "Chicago")
customer.address_street # => "May Street"
customer.address_city   # => "Chicago"
```

参考一下 has_one，让 composed_of 变得容易理解。使用 has_one 和 composed_of，两张表均属于一对一关系，只不过 composed_of 是我们想像出来的表，并不存在眞正的数据库里。

> Note: composed_of 创建的是'值对象'，区别于一般的'实体对象'。值对象没有唯一身份标识，只有所有的值相等，两个值对象才相等；而实体对象，有唯一标识(如：id)，只要唯一标识相等，两个实体对象就相等了。

可选参数详解

参数	解释
class_name	和其它关联一样，可以指定类名
mapping	旧字段与新字段的映射关系 旧字段在前，新字段在后
allow_nil	被关联的对象，其属性全部为空时，这个被关联的对象是否为 nil 对象 默认选项为 false
constructor	如何初始化值对象
converter	给值对象赋值时，如何处理

```
# 默认 allow_nil: false
customer composed_of :address, allow_nil: false
# 则有
customer = Customer.new
customer.address # 为 Address 对象，非 nil

# 设置 allow_nil: true
customer composed_of :address, allow_nil: true
# 则有
customer = Customer.new
customer.address # 为 NilClass 对象，为 nil

# 不建议使用 constructor 和 converter，而是用"类"来代替
# 因为测试、维护都不方便，特别是有一定复杂度的时候
composed_of :ip_address,
            class_name: 'IPAddr',
            mapping: %w(ip to_i),
            constructor: Proc.new { |ip| IPAddr.new(ip, Socket::
AF_INET) },
            converter: Proc.new { |ip| ip.is_a?(Integer) ? IPAdd
r.new(ip, \n
                                           Socket::AF_INET) : IPAddr
.new(ip.to_s) }

constructor # 第一次调用 x.ip_address 如何初始化
converter   # 调用 x.ip_address= 时，如何处理值对象
```

默认 关联对象只有在第一次调用，才会初始化 注意这个特点，否则你会发再一些奇怪现象。例如：

```
customer = Customer.new

# 第一次调用，初始化关联对象
customer.address
=> #<Address:0x007f8dabd87940 @street=nil, @city=nil>

# 赋值
customer.address_street = "Hyancintvej"
customer.address_city   = "Copenhagen"

# 奇怪现象
# customer 的 address 关联对象之前已经被实例化了，所以上面赋值"不起作用"。

customer.address        # => #<Address:0x007f8dabd87940 @street=
nil, @city=nil>

customer.save
customer.reload

# 保存、重新加载，发现上面赋值"已经起作用"。
customer.address
=> #<Address:0x007fe99a433a60 @city="Copenhagen", @street="Hyanc
intvej">
```

Builder - 功能的实现

实际指的是 Builder Associations，关联方法带来的基本方法。

主要做 6 件事

```
# 实现扩展，调用时传递的 block (一对多、多对多时才可用)
define_extensions model, name, &block

# 关联两者
create_reflection model, name, scope, options, extension

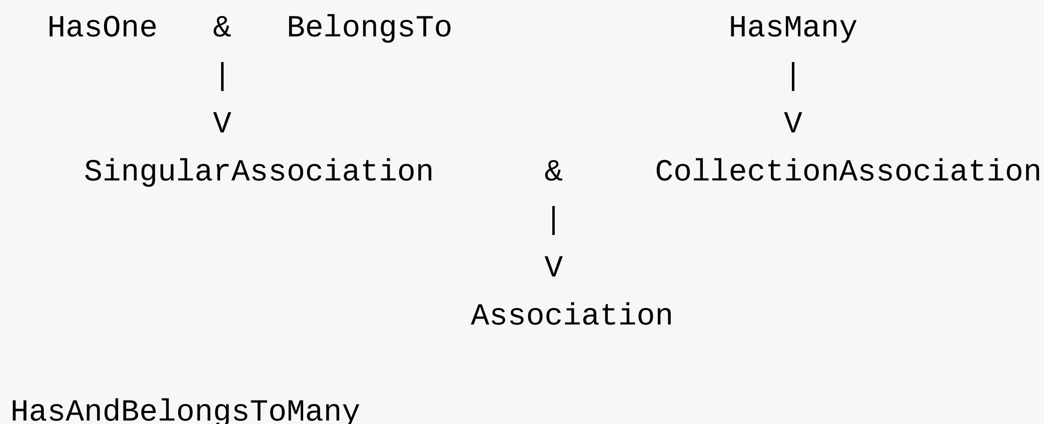
# 和关联有关的读写访问器
define_accessors model, reflection

# 和关联有关的回调(删除、自动保存等)
define_callbacks model, reflection

# 和关联有关的校验
define_validations model, reflection

# 对关联对象(特别是对象集合)的增删查改
```

关系图



```
  HasOne   &   BelongsTo              HasMany
           |                             |
           V                             V
    SingularAssociation     &     CollectionAssociation
                            |
                            V
                       Association

HasAndBelongsToMany
```

其它

- 参数相关处理：
    - 参数是否合法
    - 中间表(join_table, class_name, source 及多对多关系的自动生成...等)
    - autosave 相关实现

> Note: 对关联表的处理时，大量使用了 reflection 里面的实例方法。

~~1) Association~~

类方法：

```
class << self
  attr_accessor :extensions
  attr_accessor :valid_options
end

build

create_builder

define_callbacks
define_accessors
define_readers
define_writers
define_validations

valid_dependent_options
```

实例方法：

```
attr_reader :name, :scope, :options

build

macro

valid_options

validate_options

define_extensions
```

其它类方法：

```
check_dependent_options

add_destroy_callbacks
```

~~2) Singular Association~~

类方法：

```
define_accessors
define_constructors
define_validations
```

实例方法：

```
valid_options
```

~~3) Collection Association~~

类方法：

```
define_callbacks
define_callback

define_readers
define_writers
```

实例方法：

```
valid_options

attr_reader :block_extension

define_extensions
```

其它实例方法：

```
wrap_scope
```

~~4) Has One~~

类方法：

```
valid_dependent_options
```

其它类方法：

```
add_destroy_callbacks
```

实例方法：

```
macro
valid_options
```

~~5) Belongs To~~

类方法：

```
valid_dependent_options

define_callbacks
define_accessors
```

实例方法：

```
macro
valid_options
```

其它类方法：

```
add_counter_cache_methods
add_counter_cache_callbacks

touch_record

add_touch_callbacks
add_destroy_callbacks
```

~~6) Has Many~~

类方法：

```
valid_dependent_options
```

实例方法：

```
macro
valid_options
```

~~7) Has And Belongs To Many~~

实例方法：

```
attr_reader :lhs_model, :association_name, :options

through_model
middle_reflection
```

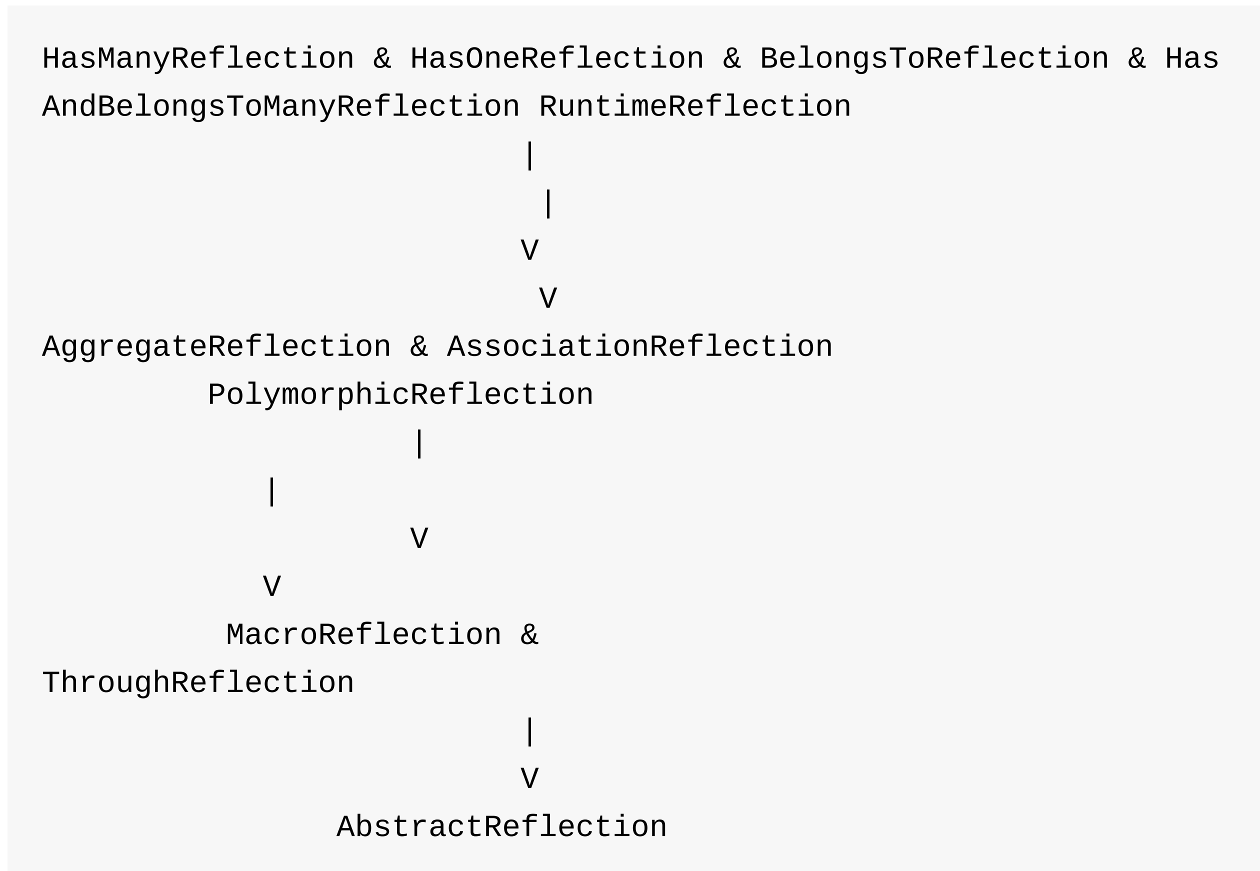
其它实例方法：

```
middle_options

belongs_to_options
```

Reflection - 实现之关联两者

关系图



```
HasManyReflection & HasOneReflection & BelongsToReflection & Has
AndBelongsToManyReflection RuntimeReflection
                          |
                           |
                          V
                           V
AggregateReflection & AssociationReflection
         PolymorphicReflection
                    |
            |
                    V
            V
          MacroReflection &
ThroughReflection
                          |
                          V
               AbstractReflection
```

延续 **build** 目录下 **association** 文件的工作。

一个很重要的概念，包含了所有的关联信息。 包括但不限于：用的是什么关联、关联对象名字、可选参数等。

一般关联和 aggregate 要区分开来。前者用 _reflections，后者用 aggregate_reflections.

提供方法：

1) 模块方法

```
create

add_reflection
add_aggregate_reflection
```

`create`
可以创建 Aggregate Reflection，Has Many Reflection、Has One Reflection 和 Belongs To Reflection 4 种关联；
如果使用了 :through 则还会自动生成 Through Reflection 关联。

2) 类方法

```
reflections # 所有正常的关联
reflect_on_all_associations          # 指定 macro 的 reflections
reflect_on_all_autosave_associations # 包含 autosave: true 的 reflections

reflect_on_all_aggregations # 所有 aggregate 关联

# 以下两方法要提供被关联对象名字
reflect_on_association
reflect_on_aggregation
```

使用举例

```
User.reflections.keys
=> [:comments,
 :warehouse]

User.reflections.each_pair { |a, x| puts [a, x.macro].join(' => '
) };
=> comments  => has_many
   warehouse => belongs_to

User.reflections.values.first.class
=> ActiveRecord::Reflection::AssociationReflection

r = User.reflections[:warehouse]
=> #<ActiveRecord::Reflection::AssociationReflection:0x007ff4606
c66d0
 @active_record=
  User(id: integer, login: string, email: string, warehouse_id:
integer),
 @class_name="Warehouse",
 @collection=false,
 @klass=
  Warehouse(id: integer, name: string),
 @macro=:belongs_to,
 @name=:warehouse,
 @options={},
 @plural_name="warehouses",
 @quoted_table_name="`warehouses`",
 @table_name="warehouses">

r.active_record
=> User(id: integer, login: string, email: string, warehouse_id:
 integer)

Post.reflections[:comments].table_name       # => "comments"
Post.reflections[:comments].macro            # => :has_many
Post.reflections[:comments].primary_key_name # => "message_id"
Post.reflections[:comments].foreign_key      # => "message_id"
```

其它

Reflection 虽然很重要，但对于普通 Web 开发者而言，使用场景有限，一般不会直接使用。下面是我想到的一些使用场景，供参考：

1. 动态创建其关联对象的实例，如：在表单里点击按钮，创建一个嵌套对象(属性)。
2. 查看使用 gem 后引进了什么关联。
3. 删除某个重要对象时，删除所有与之关联的对象。预防用 :dependent 或手动删除会有遗漏。

~~Abstract Reflection~~

```
build_association

table_name
quoted_table_name
primary_key_type
class_name
join_keys
```

~~Macro Reflection~~

继承于 Abstract Reflection

```
attr_reader :name, :scope, :options, :active_record, :plural_nam
e

autosave=
klass
compute_class
==
```

~~Association Reflection~~

Association Reflection 继承于 Macro Reflection 又继承于 Abstract Reflection

```
klass
compute_class

attr_reader :type, :foreign_type
attr_accessor :parent_reflection

association_scope_cache

constructable?

join_table
foreign_key

association_foreign_key
association_primary_key

active_record_primary_key

counter_cache_column

check_validity!
check_validity_of_inverse!
check_preloadable! & check_eager_loadable!

join_id_for

through_reflection
source_reflection

chain

scope_chain
inverse_of
polymorphic_inverse_of
macro
```

```
association_class

nested?
has_inverse?
collection?
validate?
belongs_to?
has_one?
polymorphic?
```

举例：可选参数 :inverse_of 可与哪些关联或不可与哪些可选参数一起使用。

```
VALID_AUTOMATIC_INVERSE_MACROS = [:has_many, :has_one, :belongs_
to]
INVALID_AUTOMATIC_INVERSE_OPTIONS = [:conditions, :through, :pol
ymorphic, :foreign_key]
```

另，Has Many Reflection、 Has One Reflection、Belongs To Reflection 和 Has And Belongs To Many Reflection 都继承于 Association Reflection. 它们这几个方法比较少，就不再一一列举。

~~Aggregate Reflection~~

继承于 Macro Reflection

```
mapping
```

~~Has Many Reflection~~

继承于 Association Reflection

```
macro

collection?
```

~~Has One Reflection~~

继承于 Association Reflection

```
macro

has_one?
```

~~Belongs To Reflection~~

继承于 Association Reflection

```
macro

belongs_to?

join_keys
join_id_for
```

~~Has And Belongs To Many Reflection~~

继承于 Association Reflection

```
macro

collection?
```

~~Through Reflection~~

继承于 Abstract Reflection

```
attr_reader :delegate_reflection

delegate :foreign_key, :foreign_type, :association_foreign_key,
         :active_record_primary_key, :type, :to => :source_refle
ction

klass
source_reflection
through_reflection
chain
scope_chain
join_keys

nested?
association_primary_key
source_reflection_names
source_reflection_name
source_options
through_options
join_id_for
check_validity!
```

~~Polymorphic Reflection~~

```
klass
scope

table_name
plural_name

join_keys

type

constraints

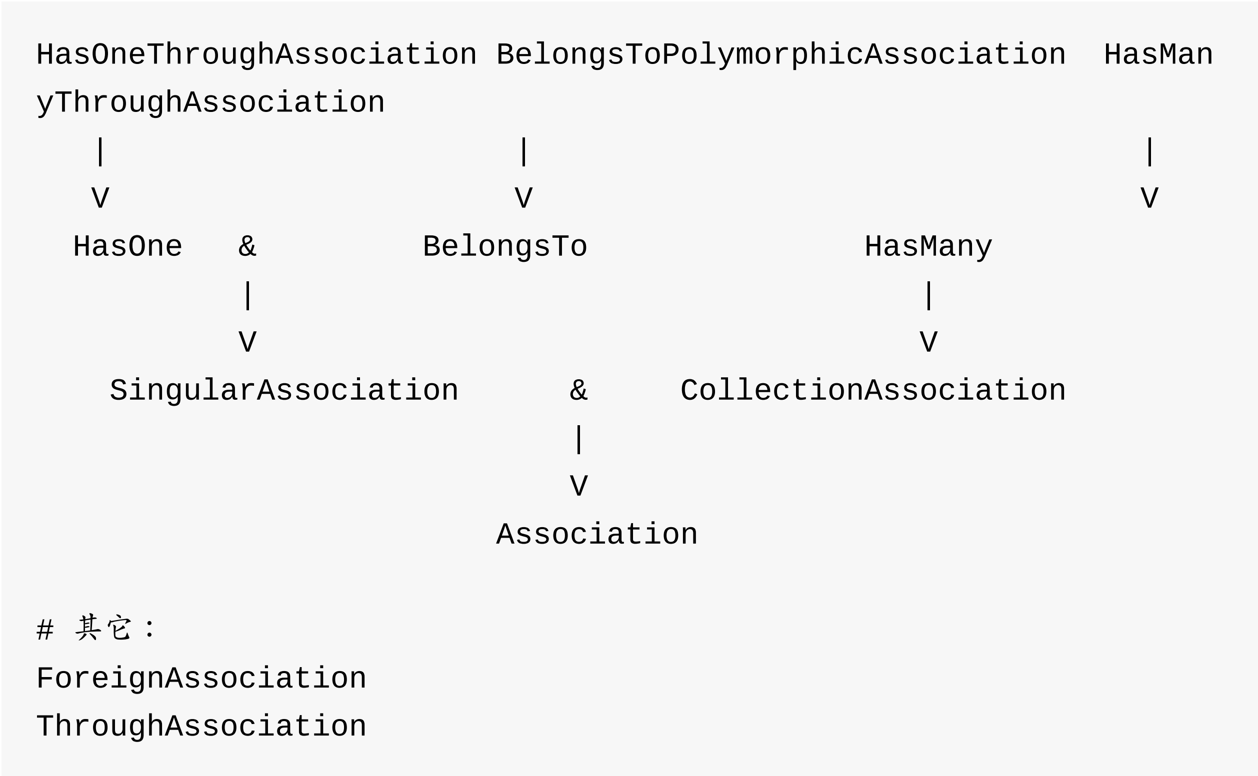
source_type_info
```

~~Runtime Reflection~~

```
klass
table_name
constraints
source_type_info
alias_candidate
alias_name
all_includes
```

Association 目录 - 实现之提供方法

关系图



```
HasOneThroughAssociation BelongsToPolymorphicAssociation  HasManyThroughAssociation
   |                       |                                 |
   V                       V                                 V
  HasOne    &          BelongsTo               HasMany
            |                                     |
            V                                     V
    SingularAssociation      &     CollectionAssociation
                             |
                             V
                        Association

# 其它:
ForeignAssociation
ThroughAssociation
```

关联方法带来的高级的方法，对外提供接口。

概念：a.b 或 a.bs 组成一个 Association，把它们看成是一个整体。但前者为 owner，后者为 target.

实现对关联对象的操作。

~~Association~~

```
attr_reader :owner, :target, :reflection
attr_accessor :inversed

delegate :options, :to => :reflection
```

```
aliased_table_name

reset

reload
loaded?
loaded!

stale_target?

target=

scope

association_scope

reset_scope

set_inverse_instance

klass

target_scope

load_target

interpolate

marshal_dump
marshal_load

initialize_attributes
```

调用到了 AssociationScope、AssociationRelation 等类。

~~Belongs To Association~~

```
handle_dependency

replace
reset

updated?

decrement_counters
increment_counters
```

~~Belongs To Polymorphic Association~~

```
klass
```

~~Collection Association~~

```
reader
writer

ids_reader
ids_writer

reset
select

find

first
second
third
fourth
fifth
forty_two
last

build
create
create!

concat

transaction

delete_all
destroy_all

count

delete
destroy

size
length
```

```
empty?
any?
many?

distinct
uniq

replace

include?

load_target
add_to_target
replace_on_target

scope
null_scope?
```

~~Foreign Association~~

```
foreign_key_present?
```

~~Has Many Association~~

```
handle_dependency

insert_record

empty?
```

~~Has Many Through Association~~

```
size
concat

concat_records
insert_record
```

~~Has One Association~~

```
handle_dependency

replace
delete
```

~~Has One Through Association~~

```
# 直接替换关联对象的话，如果创建、删除
replace

# 有关联对象及设置 :dependent 时，3 种删除方式
delete

# 处理依赖
handle_dependency
```

~~Singular Association~~

```
reader
writer

create
create!

build
```

~~Through Association~~

```
delegate :source_reflection, :through_reflection, :to => :reflec
tion
```

4 个关联方法的使用

对外提供接口。

Rails 提供了 4 个类方法，用于声明对象与对象之间的关系。它们的意思，理解起来很简单，和字面意思一样，如"Project has one Project Manager" 或 "Project belongs to a Portfolio". 但实际使用过程，还是有很多要注意的。不同的场景，需要不同的参数；并且，它们会引入一些额外的方法，有的和 attr_* 类似，但有的不是。

```
class Project < ActiveRecord::Base
  belongs_to              :portfolio
  has_one                 :project_manager
  has_many                :milestones
  has_and_belongs_to_many :categories
end
```

上面例子里的场景最简单，不需要带任何参数；下面是它们引入的额外的方法，包括但不限于：

```
# 1
Project#portfolio
Project#portfolio=(portfolio)
Project#portfolio.nil?

# 2
Project#project_manager
Project#project_manager=(project_manager)
Project#project_manager.nil?,

# 3
Project#milestones.empty?
Project#milestones.size
Project#milestones
Project#milestones<<(milestone)
Project#milestones.delete(milestone)
Project#milestones.destroy(milestone)
Project#milestones.find(milestone_id)
Project#milestones.build
Project#milestones.create

# 4
Project#categories.empty?
Project#categories.size
Project#categories
Project#categories<<(category1)
Project#categories.delete(category1)
Project#categories.destroy(category1)
```

其它

注意：可选参数 :dependent 在 has_one 和 belongs_to 里，删除关联用的是 delete；而在 has_many 里，删除关联用的是 delete_all. 这两者要做的事性质上是一样的，但请注意这点区别，并且它们都没有 destroy_all.

上述对方法或参数的解释，仅供参考，实际情况以运行结果为准。

当不确定参数表示什么意思，使用后有什么效果时，请慎用，可以选择其它确定/有把握的方法代替。

belongs_to

方法本身表示什么意思。

指定一对一关系。如果，"自己 belongs_to 关联对象"，那么自己需要包括"关联对象的外键"。

引进了哪些方法，表示什么意思。

...

后面几个方法，作用类似，只是细节部分有所不同，使用时注意一下即可。

有什么参数，表示什么意思，使用后有什么效果。

普通参数 **Scope**

设置一个 scope，通过前者查询后者的时候(其它时候不影响)，自动加到查询语句里。(类似在后者 model 里定义了一个 default_scope，但只有通过前者查询才起作用)

举例:

```
# 关联对象的 id = 2
belongs_to :user, -> { where(id: 2) }
# 同时 join 关联对象的 friends
belongs_to :user, -> { joins(:friends) }
# 关联对象的 game_level 必须大于 level.current
belongs_to :level, ->(level) { where("game_level > ?", level.current) }
```

其它

使用 primary_key 前后对比：

```
author belongs_to :book

author.book
# => SELECT `books`.* FROM `books` WHERE `books`.`id` = author.i
d LIMIT 1

author belongs_to :book, primary_key: :alias_book_id

author.book
# => SELECT `books`.* FROM `books` WHERE `books`.`alias_book_id`
 = author.id LIMIT 1
```

has_one

方法本身表示什么意思。

指定一对一关系。如果，"前者 has_one 后者"，那么'后者'需要包括'前者_id'属性。

引进了哪些方法，表示什么意思。

...

这里的解释参考了 `belongs_to` ，不单它们的方法名是一样的。定们定义方法就也是一样的。

另，belongs_to 和 has_one 都属于 Singular Association.

有什么参数，表示什么意思，使用后有什么效果。

普通参数 **Scope**

设置一个 scope，通过前者查询后者的时候(其它时候不影响)，自动加到查询语句里。(类似在后者 model 里定义了一个 default_scope，但只有通过前者查询才起作用)

举例:

```
has_one :author, -> { where(comment_id: 1) }
has_one :employer, -> { joins(:company) }
has_one :dob, ->(dob) { where("Date.new(2000, 01, 01) > ?", dob)
 }
```

其它

使用 primary_key 前后对比：

```
book has_one :author

book.author
# 传递的值是 book.id
# => Author Load (6.5ms) SELECT `authors`.* FROM `authors` \n
                         WHERE `authors`.`book_id` = book.id LIM
IT 1

book has_one :author, primary_key: :a_primary_id

book.author
# 传递的值是 book.a_primary_id
# => Author Load (6.5ms) SELECT `authors`.* FROM `authors` \n
                         WHERE `authors`.`book_id` = book.a_prim
ary_id LIMIT 1
```

has_many

方法本身表示什么意思。

指定一对多关系。

引进了哪些方法，表示什么意思。

...

以上解释，参考了官方文档。但因为各个参数组合起来，呈几何倍数的增长，验证过程难免有疏漏，实际情况以运行结果为准。

有什么参数，表示什么意思，使用后有什么效果。

普通参数 Scope

设置一个 scope，通过前者查询后者的时候(其它时候不影响)，自动加到查询语句里。(类似在后者 model 里定义了一个 default_scope，但只有通过前者查询才起作用)

举例:

```
# scope 里面的查询对象默认是被关联对象
has_many :comments, -> { where(author_id: 1) }
has_many :employees, -> { joins(:address) }

# scope 里面的查询对象用到自己
has_many :posts, ->(post) { where("max_post_length > ?", post.le
ngth) }
```

普通参数 Extension

前面提到过，使用这几个关联方法时，"它们会引入一些额外的方法"，并且上文已经介绍了它们。但如果我们想要引入自己的方法，并且用途基本一样，要怎么做，可以用 Extension.

举例:

```
class Organization < ActiveRecord::Base
  has_many :people do
    def find_active
      find(:all, :conditions => ["active = ?", true])
    end
  end
end
```

现在，可以用 `organization.people.find_active` 对后者进行操作。这样定义的方法，和 CollectionProxy 定义的方法类似。

> Note: 这些 Extension 方法，类似在后者 model 里定义的 scope, 但只能通过前者.后者这种方法调用！

has_and_belongs_to_many

方法本身表示什么意思。

指定多对多关系。用中间表来连接前者与后者。如果没有明确的指定中间表的名字，那么默认按前者、后者的字母顺序排序构成中间表。

引进了哪些方法，表示什么意思。

...

有什么参数，表示什么意思，使用后有什么效果。

普通参数 Scope

设置一个 scope，通过前者查询后者的时候(其它时候不影响)，自动加到查询语句里。(类似在后者 model 里定义了一个 default_scope，但只有通过前者查询才起作用)

举例:

```
has_and_belongs_to_many :projects, -> { includes :milestones, :m
anager }
has_and_belongs_to_many :categories, ->(category) {
  where("default_category = ?", category.name)
}
```

普通参数 Extension

前面提到过，使用这几个关联方法时，"它们会引入一些额外的方法"，并且上文已经介绍了它们。但如果我们想要引入自己的方法，并且用途基本一样，要怎么做，可以用 Extension.

举例:

```
has_and_belongs_to_many :contractors do
  def find_or_create_by_name(name)
    first_name, last_name = name.split(" ", 2)
    find_or_create_by(first_name: first_name, last_name: last_na
me)
  end
end
```

现在，可以用 `organization.people.find_active` 对后者进行操作。这样定义的方法，和 CollectionProxy 定义的方法类似。

> Note: 这些 Extension 方法，类似在后者 model 里定义的 scope, 但只能通过前者.后者这种方法调用！

关联方法的可选参数汇总

belongs_to

valid_options

```
ActiveRecord::Associations::Builder::BelongsTo.valid_options nil
```

valid_dependent_options

```
ActiveRecord::Associations::Builder::BelongsTo.valid_dependent_o
ptions
```

has_one

valid_options

```
ActiveRecord::Associations::Builder::HasOne.valid_options throug
h: 'fakers'
```

valid_dependent_options

```
ActiveRecord::Associations::Builder::HasOne.valid_dependent_opti
ons
```

has_many

valid_options

```
ActiveRecord::Associations::Builder::HasMany.valid_options nil
```

valid_dependent_options

```
ActiveRecord::Associations::Builder::HasMany.valid_dependent_opt
ions
```

has_and_belongs_to_many

和 has_many 一样。

实现关联对象：

参数	belongs_to	has_one	has_many	habtm
:class_name	√	√	√	√
:foreign_key	√	√	√	√
:foreign_type	√	√	√	√
:primary_key	√	√	√	√
:as		√	√	√
:through		√	√	√
:source		√	√	√
:source_type		√	√	√
:join_table			√	√
:association_foreign_key				√
:table_name			√	√
:autosave	√	√	√	√
:dependent	√	√	√	√
:before_add			√	√
:after_add			√	√
:before_remove			√	√
:after_remove			√	√
:validate	√	√	√	√
:required	√	√		
:counter_cache	√		√	√
:polymorphic	√			
:touch	√			
:inverse_of	√	√	√	√
:anonymous_class	√	√	√	√
:optional	√			
:extend			√	√
:index_errors			√	√

详解：

class_name

```
指定关联对象所对应的"类/Model"，默认是"驼峰"转换关联对象的名字。
不符合约定时，用 :class_name 指明。

:class_name 不影响 :foreign_key "关联对象_id"属性的命名约定。
```

foreign_key

```
belongs_to - 默认是"关联对象_id"，外键存在在自己表里。

has_many 等 - 声明自己在关联对象里的外键。
```

primary_key

```
查询关联对象的时候，用自己的哪个字段做为条件。
指定关联对象的"主键"，保存在关联对象的表里，查询关联对象时用到。查询关联对象时，自己保存的外键对应着关联对象的主键(默认是 id)，如果不符合约定，可以用 :primary_key 指明。

凡是通过前者查询后者。传递的是前者的主键所对应的值，也就是 id 字段的值，如果觉得不合适，可以用 :primary_key 指明字段。
```

as

```
多态时用到，声明多态的接口。和 belongs_to 里的 :polymorphic 配对使用。
```

through

```
在中间表里，希望关联对象怎么被表示。(听起来是不是有点绕，管得也太多了吧)
指定中间表。优先级比 :class_name、:primary_key 和 :foreign_key 要高。
实现关联时用到了 reflection 的代码，所以 has_one 或 belongs_to 关联要使用中间表，只能通过 :through 这一种方式，区别于 has_many :through 和 has_and_belongs_to_many.

指定中间表。优先级比 :class_name, :primary_key 和 :foreign_key 要高。<br> 如果对应的关联是 belongs_to 则，关联表会被自动更新、创建、删除。否则，关联表为只读状态，也就是说创建后你就只能手动维护，不会自动更新、删除。 <br> 如果你要更改关联，最好配置一下 :inverse_of 选项，以便被关联对象及时更新与其父亲的关系。
```

source

```
必须和 :through 配合使用，自己在中间表里用什么表示。(类似 :as)

配合 :through 使用，当查询关联表数据时用哪张表的字段。例如 has_many :subscribers, through: :subscriptions，如果不指定，默认会查询 :subscribers 或 :subscriber 表

中间表关联着前者和后者，并且"前者.后者"可拆分成1)"前者.中间表"，2)"中间表.后者"。第 1 步一般不会有误，但如果后者名字不规范，那么在第 2 步"中间表.后者"就会走不下去。用 :source 明确后者对应中间表里的什么关联。
```

source_type

```
必须和 polymorphic 配合使用。多态时希望自己用什么做为类型。
影响的是数据，不是属性。

从后者的角度来看，后者与前者的关联应该是 belongs_to. 但如果恰好又是多态，那么后者保存有前者的 id 并指定某个类型. 如果你对按约定生成的类型不满意，可以用 :source_type 指明。
```

foreign_type

```
1 对于 has_one 和 has_many，用什么字段做为自己的外键。<br>
2 对于 belongs_to，用什么字段做为关联对象的外键，字段保存在前者。

指定用什么"字段"保存关联对象的"多态信息"，保存在自己的表里，多态时用到。默认用"关联对象_type"这个字段，不符合约定时，用 :foreign_type 指明。

没有这个选项之前，这个字段只能根据 `:as` 生成，不能自定义。

自己不符合约定。多态时，在关联对象的表里，用什么字段来存储父亲对象的类型(默认是 x_type，根据 :as 而来)

多态时，在关联对象的表里，用什么字段来存储父亲对象的类型(默认是 x_type，根据 :as 而来)
```

optional

```
belongs_to 默认会检测其关联的对象是否存在，如果不存在则不能保存自己。设置为 false 后，则没有上述特性。
```

extend

```
作用和 extension 类似。
```

join_table

```
中间表。

指定中间表的名字 **注意:** 如果你给中间表加了对应的'类'，并且命名不符合约定的话。那么一定要记得在每个 has_and_belongs_to_many 的地方都要设置 join_table

- 注意前者与后者的顺序比较方法，如果前几个字符一样，则往后一个个字符比较，如 "paper_boxes" 和 "papers" 生成的中间表名字是 "paper_boxes_papers" 而不是 "papers_paper_boxes".
- 如果前者和后者使用的表名都带有前缀并且还相同，那么中间表的名字也用同样的前缀，剩余部分用再按字母顺序排序。如："catalog_categories" 和 "catalog_products" 生成的中间表是 "catalog_categories_products".
```

association_foreign_key

```
在中间表里，希望关联对象用什么字段做为外键。(听起来是不是有点绕，管得也太多了吧)

指定后者所对应的外键，如 Person has_and_belongs_to_many :projects, 则中间表里后者所对应的外键是“project_id”。如果不符合要求，你使用使用 :association_foreign_key 设置

作用：相当于 has_and_belongs_to_many 时 belongs_to 部分的 foreign_key
```

autosave

```
保存自己时，自动保存关联对象。
如果在 model 里使用了 accepts_nested_attributes_for，则对应 :autosave 始终为 true.

设置为 true, 保存自己的时候，同时保存它的关联对象(用的是 before_save)。设置为 false 还可分为两种情况：前者为 new_record，则保存自己时会自动保存关联对象；否则上述自动操作都不会被执行。

设置为 true，保存父亲对象时，其关联对象同时被保存。设置为 false, 对关联对象不做任何操作。默认，只有被关联对象为 new_record 时才会自动保存。<br> 使用 accepts_nested_attributes_for 会自动设置 :autosave 为 true.

以 before_save 的形式来调用，所以会受到其它 before_save 回调方法的影响。
```

dependent

```
1 belongs_to 只有 destroy 和 delete<br>
2 has_many 有 destroy、delete_all、nullify、restrict_with_exception 和 restrict_with_error<br>
3 has_one  有 destroy、delete、nullify、restrict_with_exception 和 restrict_with_error

  - `:destroy` 删除(destroy)所有被关联对象。

  - `:delete_all` 和 destroy 类似，也是删除所有被关联对象。但区别在于，此删除操作不会触发回调。

  - `:nullify` 设置后者的"前者_id"属性为 nil. 不会触发回调。

  - `:restrict_with_exception` 有关联对象则抛异常。并且后面与之的无关代码也不能再运行。

  - `:restrict_with_error` 有关联对象则设置对象的 errors 信息。并且后面与之无关的代码还能运行。

如果设置为 :destroy, 自己被 destroy 时，关联对象会被 destroy. 如果设置为 :delete, 自己被 destroy 时，关联对象会被 delete. 注意：自己被 delete, 始终不影响关联对象。

删除前者时，对后者进行什么操作。1) :destroy, 删除后者，会触发回调；2) :delete, 删除后者，不会触发回调；3) ::nullify, 把后者里的"前者_id"属性设置为 nil, 不会触发回调；4) :restrict_with_exception, 如果有后者的关联对象，报异常；5) :restrict_with_error 如果有后者的关联对象，报错

可选 :destroy，也就是使用 destroy 删除所有关联对象；可选 :delete_all，也就是使用 delete 删除所有关联对象；可选 :nullify，把外键设为 nil，但不删除对象；可选 :restrict_with_exception，有关联对象则抛异常；可选 :restrict_with_error，有关联对象则抛错误

当关联对象与自己的关系是 has_many 时，请慎用 `:dependent`. 因为关联对象被同时删除的话，意味着自己的兄弟将成为孤儿(没有关联对象可关联)。
```

validate

```
保存自己的时候，校验内存里的关联对象(不是相应字段)是否存在于数据库里。

1 设置了 validate<br>
2 不设置 validate，但设置了 autosave, 或 has_many 和 habtm 关联。<br
>
3 方法名是：autosave_association 里的 :"validate_associated_record
s_for_#{关联对象}"<br>
4 内容是 :validate_collection_association 或 :validate_single_ass
ociation<br>
5 后面使用 validate 进行执行<br>
6 校验关联是否成立，并且关联对象是否存在。

类似 validate_presence_of :关联对象

设置为 true, 保存自己的时候，会先校验它的关联对象。默认是 false, 也就是不
校验。

对于是否是 new_record 在做 valid? 时，会造成迷惑。牢记，`:validate` 意
味着对前者和后者都有"保存"操作。如果校验失败，则本次保存失败。

校验关联对象是否真实存在于数据库里。设置为 false, 保存前者时不会校验后者。
默认就是 false.
```

inverse_of

```
通过自己查找到关联对象，然后又通过关联对象找回自己。

有的关联会自动推断 inverse_of，所以用不用其实效果一样。而有的关联加上参数后
，不起作用，所以即使设置也没用。可以根据以下代码进行检测：

ModelName.reflections.map do |key, value|
  p "#{key} inverse_of: #{value.has_inverse?}"
end
```

counter_cache

```
分为几种情况：

1 belongs_to 里设置为 true 表明使用 counter. (字段使用默认)<br>
2 belongs_to 里设置 counter 字段。<br>
3 has_many   里设置 counter 字段。

设置为 true 后，自己被创建或删除，会改变关联对象里"计数器的值"。默认是 fal
se，也就是不起作用。

计数器是根据"前者对应的 table_name + count"生成的，如里不符合"约定"，可
以用 :counter_cache 指明。自定义计数器名字的时候，建议把此属性声明为 attr
_readonly. 综上，:counter_cache 可以设置为 true、false、或自定义的名字
。

定制用什么字段保存关联表的统计数目
```

touch

```
1 设置为 true，更新自己后，更新关联对象的 updated_at/on 字段。<br>
2 设置为其它字段，则更新自己后，额外更新关联对象的 updated_at/on 和其它字
段。

设置为 true, 保存或 destroy 自己的时候，关联对象的 updated_at/on 属性会
被更新。

如果你不是设置成 true, 而是传递一个符号 :symbol，那么这个符号会被更新为当
前时间。
```

required

```
要求有关联对象存在于数据库。

语法糖。原来的做法是 "belongs_to 关联对象" + "validates_presence_of
关联对象"，现改为 "belongs_to 关联对象, required: true"

"validates_presence_of 关联对象" 和 "validates_presence_of 关联对象
_id" 大同小异。前者是真实存在的对象，后者只是字段。
```

readonly

```
所有关联对象为只读。

设置为 true，通过前者查询到的后者限制为只读状态，不可更改。但其它方式查询出
来的，不受此限制。
```

polymorphic

```
声明此关联是多态的

如果你同时使用了 `:counter_cache`，建议在后者的 model 里把计数器设置为 a
ttr_readonly.
```

inverse_of

```
保证 object_id 相同。通过前者(1)查询到后者，然后再通过后者返过来查询前者(2
)。按照直观的理解，(1) 和 (2) 应该是同样的对象，同样的值。但实际情况会发现
，它们不一样(可以通过 object_id 确定)！原因是程序没有这么聪明，没法判断它
们是一样的(特别是通过中间表查询时)。设置 :inverse_of，可解决这个问题。

带来额外的好处：1. 不用重复查询，节省了性能；2. 因为对象的 object_id 都是
一样的，保证了数据一致性；3. 注意顺序前者查询后者，然后后者反向查询前者；需
要注意：第 2 次"查询"不是数据库查询，会存在操作脏数据的风险。

通常情况下，不用设置，会自动转换。但使用了以下参数，则不会自动转换：:throug
h、:as、:polymorphic 和 :conditions；遇到单复数不规则，有时候也不会自动
转换
```

index_errors

```
默认为 false. 设置了 autosave 它才管用。条件很复杂，忽略。
```

anonymous_class

```
Reflection 那边用到，忽略。
```

before_add、after_add、before_remove 和 after_remove

> 作用于集合，和普通的回调差不多，只是定义的位置不同而矣。

心得

关系比较复杂的时候，不好写。我建议先从简单、可确定的入手，然后进行下一步。

例如：自关联，或者通过关联表实现关联，通常：

- 表里有 `xxx_id` 等外键的，通常就是 `belongs_to`
- 然后对应其关联的就是 `has_many`
- 再之后就是 `through`

4 个关联方法的补充

注意事项

只有 :has_many, :has_one, :belongs_to 才有可能自动计算 inverse，也就是说 has_and_belongs_to_many 不可以。

若上述声明里包含 :conditions, :through, :polymorphic, :foreign_key 则同样不可以自动计算 inverse.

所有参数，目标总结起来就这几个

```
# 实现关联对象

# 调用时传递的 block
define_extensions model, name, &block

# 关联两者
create_reflection model, name, scope, options, extension

# 和关联有关的读写访问器
define_accessors model, reflection

# 和关联有关的回调(删除、自动保存等)
define_callbacks model, reflection

# 和关联有关的校验
define_validations model, reflection

# 对关联对象(特别是对象集合)的增删查改
```

使用关联方法后，Rails 自动生成了哪些方法

一对一关系，和关联有关的读写访问器

生成的方法	**belongs_to**	**belongs_to :polymorphic**	**has_one**
association(force_reload = false)	√	√	√
association=(associate)	√	√	√
build_association(attributes={})	√		√
create_association(attributes={})	√		√
create_association!(attributes={})	√		√

这些方法主要由 Association 和 Singular Association 提供。

一对多、多对多关系，和关联有关的读写访问器

生成的方法	**habtm**	**has_many**	**has_many :through**
collection(force_reload = false)	√	√	√
collection=objects	√	√	√
collection_singular_ids	√	√	√
collection_singular_ids=ids	√	√	√

这些方法主要由 Association 和 Collection Association 提供。

一对多、多对多关系，对关联对象(特别是对象集合)的增删查改

生成的方法	habtm	has_many	has_many :through
collection<<(object, …)	√	√	√
collection.delete(object, …)	√	√	√
collection.destroy(object, …)	√	√	√
collection.clear	√	√	√
collection.empty?	√	√	√
collection.size	√	√	√
collection.find(…)	√	√	√
collection.exists?(…)	√	√	√
collection.build(attributes = {}, …)	√	√	√
collection.create(attributes = {})	√	√	√
collection.create!(attributes = {})	√	√	√

上述方法都是 Collection Proxy 提供的。(除上述方法外，还有的未列出)

has_and_belongs_to_many vs has_many through

has_and_belongs_to_many 意味着中间表没有 model. 因此，不能通过 id 进行查询，也没有 validate、callback，更加不能直接读取&设置其值。唯一的方便之处是：通过 `<<` 即可创建中间表数据。

当你需要 model、id、validate、callback 或对中间表数据进行操作时，建议改为使用 has_many :throght，这时可以通过 model 管理中间表数据，不必也不能再 `<<`

关联	has_and_belongs_to_many	has_many :through
AKA	habtm	through association
Structure	Join Table	Join Model
Primary Key	no	yes
Rich Association	no	yes
Proxy Collection	yes	no
Distinct Selection	yes	yes
Self-Referential	yes	yes
Eager Loading	yes	yes
Polymorphism	no	yes
N-way Joins	no	yes

其它

TODO

~~Autosave Association~~

负责处理自动保存相关一切任务。

在 Builder::Association 里有构建：

```
def self.define_callbacks(model, reflection)
  if dependent = reflection.options[:dependent]
    check_dependent_options(dependent)
    add_destroy_callbacks(model, reflection)
  end

  Association.extensions.each do |extension|
    # 这里构建
    extension.build model, reflection
  end
end
```

而 AssociationBuilderExtension 正是 Associations::Builder::Association.extensions 其中之一。

对应着的私有类方法：

```
# 给指定的关联添加 autosave
add_autosave_association_callbacks
```

就是上面这方法添加的回调，非常重要。

常用实例方法：

```
mark_for_destruction

marked_for_destruction?
```

除此之外，还有实例方法：

```
changed_for_autosave?

destroyed_by_association
destroyed_by_association=

reload
```

私有类方法：

```
define_non_cyclic_method
```

~~Alias Tracker~~

作用：为 Join Dependency 和 Association Scope 服务。

实现对不规则表名的跟踪及相关处理。

实例方法：

```
attr_reader :aliases, :connection

aliased_table_for
```

类方法：

```
empty

create

initial_count_for
```

~~Association Scope~~

场景"关联的时候，关联表带有 scope"，不使用它的话，生成的 SQL 里能看到有的条件是重复的。Association Scope 解决了这个问题。

另，看示例还有另外的好处：

```
class Review < ActiveRecord::Base
  belongs_to :restaurant

  scope :positive, -> { where("rating > 3.0") }
end

class Restaurant < ActiveRecord::Base
  has_many :reviews
  has_many :postitive_reviews, -> { positive }, class_name: "Review"
end
```

```
restaurants = Restaurant.includes(:positive_reviews).first(5)
```

Collection Association、Singular Associations 和 Association 调用了它。

~~Join Dependency~~

作用：为 Finder Methods 和 Query Methods 服务。

实现 joins

```
class Physician < ActiveRecord::Base
   has_many :appointments
   has_many :patients, through: :appointments
end
```

以下两种查询：

```
@physician.patients.to_a

Physician.joins(:appointments).to_a
```

对应的 `joins` 语句是不同的，甚至差距很大，这部分工作由 Join Dependency 判断。

~~Preloader~~

使用 includes, preload, eager_load 进行预加载时，根据查询条件不同，优先选择对性能友好的方式。

```
class Author < ActiveRecord::Base
  # columns: name, age
  has_many :books
end

class Book < ActiveRecord::Base
  # columns: title, sales, author_id
end
```

针对上述关联，执行以下两种查询：

```
Author.includes(:books).where(name: ['bell hooks', 'Homer']).to_a

Author.includes(:books).where(books: {title: 'Illiad'}).to_a
```

它们对应的 SQL 差别很大，性能差距也大。究竟要执行哪种查询，可以让 Preloader 来判断。

~~Association Relation~~

针对 Scope `merge` 操作的实现：

```
# 可以被重写(ThroughAssociation 里就这么做)
def target_scope
  AssociationRelation.create(klass, klass.arel_table, klass.pred
icate_builder, self).merge!(klass.all)
end
```

Active Record 迁移

- Migration
- Schema
- Connection Adapters

包括：Schema Statements、Table Definition、Table 和 Database Statements 等。

- 其它

包括：Model Schema*、Command Recorder、~~Schema Dumper~~ 和 ~~Schema Migration~~.

- ~~Migrator~~
- ~~Schema Dumper~~

Connection Handling 连接数据库

用来建立和数据库的连接。(配置、建立连接、日志，其中的建立连接，但连接适配不是它做的。)

```
# 根据配置信息进行连接
establish_connection

# 连接信息
connection
```

通过 `establish_connection` 连接数据库，我们不必加载整个 Rails 环境，仅使用数据库操作这部分。

举例一：

```
require 'yaml'
require 'active_record'

dbconfig = YAML::load(File.open('config/database.yml'))
# 这里以 development 环境为例
ActiveRecord::Base.establish_connection(dbconfig["development"])
# ActiveRecord::Base.logger = Logger.new(STDERR)

class User < ActiveRecord::Base
  # your code here ...
end
```

举例二(立即加载基准测试脚本)：

```
require 'rubygems'
require 'faker'
require 'active_record'
require 'benchmark'

# This call creates a connection to our database.
```

```
ActiveRecord::Base.establish_connection(
  :adapter => "mysql2",
  :host => "127.0.0.1",
  :username => "root", # Note that while this is the default set
ting for MySQL,
  :password => "", # a properly secured system will have a diffe
rent MySQL
  # username and password, and if so, you'll need to
  # change these settings.
  :database => "test")

# First, set up our database...
class Category < ActiveRecord::Base
end

unless Category.table_exists?
  ActiveRecord::Schema.define do
    create_table :categories do |t|
      t.column :name, :string
    end
  end
end

Category.create(:name => 'Sara Campbell\'s Stuff')
Category.create(:name => 'Jake Moran\'s Possessions')
Category.create(:name => 'Josh\'s Items')
number_of_categories = Category.count

class Item < ActiveRecord::Base
  belongs_to :category
end

# If the table doesn't exist, we'll create it.

unless Item.table_exists?
  ActiveRecord::Schema.define do
    create_table :items do |t|
      t.column :name, :string
      t.column :category_id, :integer
    end
```

```
  end
end

puts "Loading data..."

item_count = Item.count
item_table_size = 10000

if item_count < item_table_size
  (item_table_size - item_count).times do
    Item.create!(:name => Faker.name,
                 :category_id => (1+rand(number_of_categories.to
_i)))
  end
end

puts "Running tests..."

Benchmark.bm do |x|
  [100, 1000, 10000].each do |size|
    x.report "size:#{size}, with n+1 problem" do
      @items=Item.find(:all, :limit => size)
      @items.each do |i|
        i.category
      end
    end
    x.report "size:#{size}, with :include" do
      @items=Item.find(:all, :include => :category, :limit => si
ze)
      @items.each do |i|
        i.category
      end
    end
  end
end
```

举例三：

```
require 'active_record'

ActiveRecord::Base.logger = Logger.new(STDERR)
# ActiveRecord::Base.colorize_logging = false

ActiveRecord::Base.establish_connection(
  :adapter => "sqlite3",
  :database => ":memory:"
)

ActiveRecord::Schema.define do
  create_table :albums do |table|
    table.column :title, :string
    table.column :performer, :string
  end

  create_table :tracks do |table|
    table.column :album_id, :integer
    table.column :track_number, :integer
    table.column :title, :string
  end
end

class Album < ActiveRecord::Base
  has_many :tracks
end

class Track < ActiveRecord::Base
  belongs_to :album
end
```

`connection` 使用举例：

```
ActiveRecord::Base.connection.table_exists? 'some_table'
```

另外一个使用 `execute` 方法举例：

```
ActiveRecord::Base.connection.execute "ALTER TABLE some_table CH
ANGE COLUMN..."
```

其它方法：

```
# 连接的配置信息
connection_config

# 是否已连接
connected?

connection_id
connection_id=

connection_pool

remove_connection
retrieve_connection
```

Migration 文件下的内容

重要的如：

```
run 或 migrate
     |
     v
 exec_migration
     |
     v
 up 或 down
```

```
revert
reversible
```

`revert` 内部其它方法会调用，懒人复制、粘贴，不想改名字使用。

其它：

```
check_pending!
disable_ddl_transaction!
load_schema_if_pending!

reverting?

run
migrate

exec_migration

announce
connection
copy
next_migration_number
proper_table_name
say
say_with_time
suppress_messages
table_name_options
write
```

控制台里迁移、回滚等命令

主要命令

```
rake db:migrate
rake db:rollback

rake db:migrate:up
rake db:migrate:down

rake db:migrate:redo
```

指定版本号的回滚

```
rake db:migrate:down VERSION=20141119130134
```

回滚最近几个迁移

```
rake db:rollback STEP=n
```

n 代表个数。注意：是最近几个，它们会被一起移除。

其它类似命令：

只执行指定版本号的迁移

```
rake db:migrate VERSION=20141119130134
```

只执行最近几次迁移

```
rake db:migrate STEP=n
```

回滚、然后重新执行最近几次迁移

```
rake db:migrate:redo STEP=n
```

say_with_time

使用举例：

```
def up
  ...
  say_with_time "Updating salaries..." do
    Person.all.each do |p|
      p.update_attribute :salary, SalaryCalculator.compute(p)
    end
  end
  ...
end
```

Connection Adapters 包含以下这 4 个模块，及其它多个模块，但在此略过它们 ...

Schema Statements

重要的操作或问询如下(实例方法)：

```
# index(索引)
add_index
remove_index
rename_index
index_exists?
index_name_exists?

# column(属性)
add_column
rename_column
change_column
remove_column
remove_columns
column_exists?

# table(表)
create_table
drop_table
rename_table
change_table
table_exists?

# 外键
add_foreign_key
remove_foreign_key
foreign_key_exists?

# 引用(区别于外键，多态也可用)
add_reference & add_belongs_to
remove_belongs_to & remove_reference
```

其它操作或问询(实例方法)：

```
add_timestamps

assume_migrated_upto_version

change_column_default
change_column_null

initialize_schema_migrations_table

create_join_table
drop_join_table

columns
foreign_keys
native_database_types

remove_timestamps

table_alias_for

view_exists?(view_name)
views
```

其它实例方法：

```
options_include_default?

add_index_sort_order
rename_column_indexes
rename_table_indexes

index_name_for_remove
quoted_columns_for_index
```

create_table

可选参数：

:id

```
create_table(:categories_suppliers, id: false) do |t|
  t.column :category_id, :integer
  t.column :supplier_id, :integer
end
```

:primary_key

```
# 改名字叫 guid
create_table(:products, primary_key: 'guid') do |t|
  # ...
end

# 主键仍然是 integer 类型
create_table :products, id: false do |t|
  t.primary_key :sku
  # ...
end
```

上述两种解决方法类似，但它们类型都是 integer ... 比如我们想要 string 呢，只能这样：

1) 使用 SQL 创建主键

```
execute "ALTER TABLE employees ADD PRIMARY KEY (emp_id);"
```

问题解决了，但另一问题又来了，这么做迁移不记录在案。如果想要记录，我们还要

```
config.active_record.schema_format = :sql
```

显然，这并不是好的做法。

另一解法是：

2) 类似主键，但不是主键

```
create_table :employees, :primary_key => :emp_id do |t|
  t.string :first_name
  t.string :last_name
end
change_column :employees, :emp_id, :string
```

上述方法，并不是真正的 primary_key，其实是 string

另外，

更改主键，则 `id` 和 `id=` 也会做相应变化，如果你觉得别扭，可以重置它们(尽管不推荐)。

```
class Product < ActiveRecord::Base
  ....
  # 很多方法都依赖于 id，不推荐这么做
  def id
    raise NoMethodError, "Please call #{self.class.primay_key} i
nstead."
  end

  def id=(value)
    raise NoMethodError, "Please call #{self.class.primay_key}=
instead."
  end
  ....
end
```

其它地方相应更改：

```
# app/models/product.rb
class Product < ActiveRecord::Base
  # model 层面指定 primay_key
  self.primay_key = 'sku'

  # 重写 to_param，更友好的 params
  def to_param
    sku.parameterize
  end
end

# config/routes.rb
# 路由里使用自定义主键代替 :id
# 在这里是 prams[:sku] 代替 params[:id]
resources :products, param: :sku
```

> Note: 总结，使用 string 类型做主键，还没有好的解决方案。

:options

```
create_table :products, options: "ENGINE=BLACKHOLE" do |t|
  t.string :name, null: false
end
```

:temporary

```
create_table(:long_query, temporary: true,
  as: "SELECT * FROM orders INNER JOIN line_items ON order_id=orders.id")
```

:force

创建表之前先删除它(如果已经存在同名表的话)，确保创建过程是顺利的。

功能类似：

```
drop_table :table_name if table_exists?("table_name")
create_table :table_name
```

:as

```
create_table(:long_query, temporary: true,
  as: "SELECT * FROM orders INNER JOIN line_items ON order_id=or
ders.id")
```

change_table

可选参数：

:bulk

尽可能的将更改表操作转换成一条 SQL 语句，一次执行。

如下面所示，一条 SQL 语句，可完成多个操作：

```
ALTER TABLE `users` ADD COLUMN age INT(11), ADD COLUMN birthdate
 DATETIME ...
```

好处是，这可以加快迁移速度。

Table Definition

简单划分 **create_table**

使用 create_table 时，传递给 block 的参数 `t` 就是此类型：

```
create_table :table_name do |t|
  # t.class
  # => ActiveRecord::ConnectionAdapters::TableDefinition
end
```

重要的如：

```
column
```

而 `t` 所支持的数据类型通过 `column` 对外开放：

其它方法：

```
remove_column

columns
index
primary_key
timestamps
belongs_to & references
```

> Note: 这里有一些方法是通过元编程生成的，查文档找不到它们，可能通过 ActiveRecord::ConnectionAdapters::TableDefinition.public_instance_methods(false) 查看。

列举如下：

```
:indexes
:indexes=
:name
:temporary
:options
:as
:columns
:primary_key
:[]
:column
:remove_column
:string
:text
:integer
:float
:decimal
:datetime
:timestamp
:time
:date
:binary
:boolean
:index
:timestamps
:references
:belongs_to
:new_column_definition
```

Table

简单划分 **change_table**

使用 create_table 时，传递给 block 的参数 `t` 就是此类型：

```
change_table :table_name do |t|
  # t.class
  # => ActiveRecord::ConnectionAdapters::Table
end
```

相关模块或方法，可以参考：TableDefinition 和 SchemaStatements#create_table

支持的数据类型或操作有:

```
change_table :table do |t|
  t.column
  t.index
  t.rename_index
  t.timestamps
  t.change
  t.change_default
  t.rename
  t.references
  t.belongs_to
  t.string
  t.text
  t.integer
  t.float
  t.decimal
  t.datetime
  t.timestamp
  t.time
  t.date
  t.binary
  t.boolean
  t.remove
  t.remove_references
  t.remove_belongs_to
  t.remove_index
  t.remove_timestamps
end
```

除上述外，还有：

```
column_exists?
index_exists?

change, change_default

remove, remove_index, remove_timestamps
remove_belongs_to & remove_references

rename, rename_index

column
index
timestamps
belongs_to & references
```

> Note: 这里有一些方法是通过元编程生成的，查文档找不到它们，可能通过 ActiveRecord::ConnectionAdapters::Table.public_instance_methods(false) 查看。

列举如下：

```
:column
:column_exists?
:index
:index_exists?
:rename_index
:timestamps
:change
:change_default
:remove
:remove_index
:remove_timestamps
:rename
:references
:belongs_to
:remove_references
:remove_belongs_to
:string
:text
:integer
:float
:decimal
:datetime
:timestamp
:time
:date
:binary
:boolean
```

Database Statements

常用的如：

```
execute
```

有时候，我们会在迁移文件里直接运行 SQL，使用上述方法，可以达到目的。

举例：

```
class MakeJoinUnique < ActiveRecord::Migration
  def up
    execute "ALTER TABLE `pages_linked_pages` \n
              ADD UNIQUE `page_id_linked_page_id` (`page_id`,`lin
ked_page_id`)"
  end

  def down
    execute "ALTER TABLE `pages_linked_pages` DROP INDEX `page_i
d_linked_page_id`"
  end
end
```

其它方法，不在此一一列举。

Model Schema

通过它们可以查看和更改 model 与 table 的对应关系。

常用类方法：

```
reset_column_information
```

```
column_names
column_defaults

inheritance_column
inheritance_column=

table_name
table_name=

table_exists?
```

在数据库层面，我们可以给某一列设置默认值。在 model 层面，我们创建对象之后，如果没有设置对应列的值，那保存的时候存的就是默认值。
那么，怎么查看所有列及其在数据库层面的默认值呢。
通过它： `column_defaults`

查看所有列的名字： `column_names`

单表继承时，默认字段为 `type` , 可以更改它： `inheritance_column=`

`reset_column_information` 定义于 Model Schema，在迁移的时候经常用到。

整个迁移过程，环境只加载一次。但有时候，我们添加了属性，我们希望马上能够填充数据。此时，就需要重新读取 model 的表信息。

举例：

```
class AddPeopleSalary < ActiveRecord::Migration
  def up
    # 添加属性
    add_column :people, :salary, :integer
    # 确保已经添加了此属性
    Person.reset_column_information
    # 马上填充数据
    Person.all.each do |p|
      p.update_attribute :salary, SalaryCalculator.compute(p)
    end
  end
end
```

其它类方法：

```
sequence_name
sequence_name=

quoted_table_name

content_columns
```

`quoted_table_name` 可用命令的形式获取 Model 对应的表名。

`sequence_name` 获取序列名。

Schema

继承于 Migration

常用的如：

```
define
```

对应的是 db/schema.rb 里的方法，举例：

```
ActiveRecord::Schema.define(version: 20380119000001) do
  # ...
end
```

再举例：

```
require 'active_record'
ActiveRecord::Base.establish_connection( adapter: 'sqlite3', database: ":memory:" )
ActiveRecord::Schema.define(version: 1) { create_table(:articles) { |t| t.string :title } }

class Article < ActiveRecord::Base; end

Article.create title: 'Quick brown fox'
```

其它方法：

```
migrations_paths
```

> Note: 它里面不支持非 migration 语句，也就是说直接让它执行 SQL 语句是不行的。

其它

TODO

schema_format

`config.active_record.schema_format` 决定生成什么 schema 文件及其内容。

设置为 :ruby (默认)则生成 `schema.rb` 文件，里面是 Ruby 代码；
设置为 :sql 则生成 `structure.sql` 文件，里面是 SQL 代码。

~~Schema Migration~~

同名方法如：

```
create_table
drop_table
```

其它：

```
index_name

normalize_migration_number

normalized_versions

primary_key

table_exists?

table_name
```

Active Record 约定、配置，底层及其它

- Model Schema
- Naming Conventions
- Schema Conventions
- Timestamps
- Core
- ~~Attribute~~

和属性的类型转换有关，在 Attribute Set 里会用到。尽管 Attribute Set 也属于底层。

- ~~Attribute Set~~
- ~~Attribute Decorators~~

相对而言较底层，使用 Rails 的话，一般不会直接用到的它们：

- Connection Handling
- Serialization
- Migration DatabaseTasks
- Statement Cache
- ~~Explain~~
- ~~Explain Registry~~
- ~~Explain Subscriber~~
- ~~Query Cache~~
- ~~Result~~
- ~~Runtime Registry~~
- ~~Sanitization~~

- ~~Legacy Yaml Adapter~~

Naming Conventions

类映射表，属性映射列。

命名约定：

Model / Class	Table / Schema
Post	posts
LineItem	line_items
Deer	deers
Mouse	mice
Person	people

Schema Conventions

命名约定：

属性	解释
Foreign keys	外键。前者 belongs_to 后者，根据后者所对应的表名，为前者生成"后者_id"属性。
Primary keys	主键。默认使用"id"属性，迁移时自动完成。

除上述属性外，约定还有:

属性	解释
created_at	创建 record 的时间戳
updated_at	最后修改 record 的时间戳
created_on	创建 record 的时间
updated_on	最后修改 record 的时间
lock_version	使用乐观锁特性的时候会用到
type	使用单表继承特性的时候会用到
(association_name)_type	使用多态特性的时候会用到
(table_name)_count	使用 counter_cache 特性的时候会用到

尽管这些属性都是可更改的，但强烈建议不要更改它们，使用默认的即可。
另外，如果没有使用到上述特性，那么强烈建议不要使用上述属性。(实在需要的话，可以找同义词替代)

Timestamps

时间戳包括 `created_at/created_on` 和 `updated_at/updated_on`

不想生成时间戳相关属性:

```
config.active_record.record_timestamps = false
```

时间戳默认使用 UTC，如果你想使用本地时区：

```
config.active_record.default_timezone = :local
```

默认，ActiveRecord 可以自动识别并转换时区：

```
config.active_record.time_zone_aware_attributes = true
```

如果你想取消此特性，可以配置为 `false`

如果你仍然想使用此特性，但同时又有需要忽略此特性，你可以手动设置，举例：

```
class Topic < ActiveRecord::Base
  self.skip_time_zone_conversion_for_attributes = [:written_on]
end
```

Core

从原来的 Base 中抽取出来单独成 module，负责部分底层对接、重写部分常用方法等。一起抽取出来的还有 Model 模块(包含 include、extend 等，现已经被删除)

包含时即执行：

```
mattr_accessor :logger, instance_writer: false

self.configurations

mattr_accessor :default_timezone, instance_writer: false
mattr_accessor :schema_format, instance_writer: false
mattr_accessor :timestamped_migrations, instance_writer: false
mattr_accessor :dump_schema_after_migration, instance_writer: fa
lse
mattr_accessor :maintain_test_schema, instance_accessor: false

class_attribute :default_connection_handler, instance_writer: fa
lse
class_attribute :find_by_statement_cache
```

实例方法：

```
<=>
== & eql?

clone
dup

connection_handler

encode_with
init_with

freeze
frozen?

hash

inspect

pretty_print

readonly!

readonly?

slice
```

`self.allocate` 和 `init_with` 使用举例：

```
class Post < ActiveRecord::Base
end

post = Post.allocate
post.init_with('attributes' => { 'title' => 'hello world' })
post.title # => 'hello world'
```

类方法一：

```
===

allocate

find
find_by
find_by!

generated_association_methods

inherited

initialize_find_by_cache
initialize_generated_modules

inspect
```

类方法二：

```
configurations
configurations=

connection_handler
connection_handler=

new
```

获取 record 对象

归结起来，有两种方式可以获得 record 对象：**1)** 从数据库里获取 **2)** 程序新创建。

前者可以通过各种方法查询得到，但都是封装了 `init_with`
后者也可以通过各种方法创建得到，但都是封装了 `initialize`

```
# 查询得到。从 SQL 层面转向到 Ruby 层面
def find_by_sql(sql, binds = [])
  result_set = connection.select_all(sanitize_sql(sql), "#{name}
 Load", binds)
  column_types = result_set.column_types.dup

  # ...

  result_set.map { |record| instantiate(record, column_types) }
end
```

`instantiate` 查询得到，等价于：`self.allocate` + `init_with`

```
def instantiate(attributes, column_types = {})
  klass = discriminate_class_for_record(attributes)
  attributes = klass.attributes_builder.build_from_database(attr
ibutes, column_types)
  klass.allocate.init_with('attributes' => attributes, 'new_reco
rd' => false)
end

def allocate
  define_attribute_methods
  super
end

def init_with(coder)
  @attributes = coder['attributes']

  init_internals

  @new_record = coder['new_record']

  self.class.define_attribute_methods

  _run_find_callbacks
  _run_initialize_callbacks

  self
end
```

`initialize` 创建得到，比如： `new` 或 `build`

```
User.new(first_name: 'Jamie')
```

```
def initialize(attributes = nil, options = {})
  @attributes = self.class._default_attributes.dup
  self.class.define_attribute_methods

  init_internals
  initialize_internals_callback

  init_attributes(attributes, options) if attributes

  yield self if block_given?
  _run_initialize_callbacks
end
```

前者，也就是查询得到，一定有 `persisted?` # => true
后者，也就是创建得到，一定有 `persisted?` # => false

其它类似：

```
clone
dup

initialize_dup
```

`clone`

```
user = User.first
new_user = user.clone
user.name               # => "Bob"
new_user.name = "Joe"
user.name               # => "Joe"

user.object_id == new_user.object_id            # => false
user.name.object_id == new_user.name.object_id  # => true

user.name.object_id == user.dup.name.object_id  # => false
```

`encode_with`

```
class Post < ActiveRecord::Base
end
coder = {}
Post.new.encode_with(coder)
coder # => {"attributes" => {"id" => nil, ... }}
```

`init_with`

```
class Post < ActiveRecord::Base
end

post = Post.allocate
post.init_with('attributes' => { 'title' => 'hello world' })
post.title # => 'hello world'
```

参考

ActiveRecord 对象的拼装
Read and Write Activerecord Attribute
ActiveRecord Object Instantiate
Using Allocate to Create ActiveRecord Objects When Stubbing find_by_sql

Serialization

有 ActiveRecord::Serializers::JSON 提供：

`serializable_hash(options = nil)` 方法。
效果和 as_json 差不多。实际起作用的是 ActiveModel::Serialization 里的同名方法。

有 ActiveRecord::Serializers::XmlSerializer 提供：

`to_xml` 方法。实现方式：include and extend ActiveModel::Serializers::Xml，并且里面有同名方法。

> Note: 包括了 XmlSerializer.
>
> Rails 5 里 ActiveRecord::Serialization::XmlSerializer 已废除。

Database Tasks

rake db:migrate 等迁移命令，定义在 Active Record 的 databases.rake 里。 其中，大部分命令都是由 ActiveRecord::Tasks::DatabaseTasks 来处理，再或者转发到其它模块。

提供接口如下：

```
create_current    # 对应 rake db:create
create_all        # 对应 rake db:create:all

drop_current      # 对应 rake db:drop
drop_all          # 对应 rake db:drop:all

purge_current     # 对应 rake db:purge
purge_all         # 对应 rake db:purge:all

migrate           # 对应 rake db:migrate

charset_current   # 对应 rake db:charset
collation_current # 对应 rake db:collation
```

其它接口，不在此一一列举：

```
charset
check_schema_file
collation
create
current_config
db_dir
drop
env
fixtures_path
load_schema, load_schema_current
load_seed
migrations_paths
purge
register_task
root
seed_loader
structure_dump, structure_load
```

Statement Cache

提供 `create(connection, block = Proc.new)` 类方法：

```
cache = StatementCache.create(Book.connection) do |params|
  Book.where(name: "my book").where("author_id > 3")
end
```

通过它可以缓存某些数据库操作(和 Relation 有点类似)。

提供 `execute(params, klass, connection)` 实例方法：

```
cache.execute([], Book, Book.connection)
```

前面缓存起来的操作，对应着一个对象。通过它运行里面真正要执行的内容。

Collection Cache Key

给 Relation 集合也可以有 cache_key，例如：

```
blogs = Blog.where("id > 2");

blogs.collection_cache_key
   (0.2ms)  SELECT COUNT(*) AS size, MAX("blogs"."updated_at") A
S timestamp
   FROM "blogs" WHERE (id > 2)
# => "blogs/query-9a5d6ea830ba519707a5aaf189b9a1b1-0"
```

`ActiveRecord::Base#cache_key` 封装了它，所以直接用 `cache_key` 即可。

Attribute Mutation Tracker

是"Dirty - 跟踪对象值的变化情况"的内部组成部分之一。

Attribute Decorators

提供以下两方法：

```
decorate_attribute_type(column_name, decorator_name, &block)

decorate_matching_attribute_types(matcher, decorator_name, &bloc
k)
```

其它

- ~~type~~

包括所有支持的数据类型

- Railtie

- errors

- ~~Base~~

- ~~FixtureSet~~

fixtures/ 相关。

- ~~tasks~~

包括 Database Tasks，及 MySQL Database Tasks、PostgreSQL Database Tasks、SQLite Database Tasks.

- databases.rake (很重要)

- ~~Explain~~

explain 方法的底层实现(尽管是调用数据库的 explain...)

Active Support autoload 的类和模块

TODO

Autoload

延迟加载(自动加载)和预加载(立即加载)。

常用方法：

```
autoload
eager_autoload
```

```
eager_load!
```

使用 require 加载

```
require "some/library"
```

使用 `require` , 如果此前没有加载过，则加载并返回 true; 如果此前已经加载过，则返回 false.
(区别于 `load` ，load 每次都会加载。没有返回 true/false 的概念)

使用 require 面临的问题

require 意味着"立即加载所有"，一开始这很美好，但随着项目的成长，启动速度就会变得很慢。因为我们并没有真正使用到"所有"文件，但却需要加载。

并且，require 不是线程安全的。

使用 autoload 自动加载(延迟加载)

```
module Foo
  autoload :Bar, "path/to/bar"
end
```

使用 `autoload` ，相当于先声明要加载的模块(纪录在 @_autoloads)，真正需要的时候才会加载。

autoload 本质还是 require, 只是把"加载"细分为"声明 + 加载"。

使用 autoload 面临的问题

多线程时，这又会引出一个新问题。就是在一个线程里需要此模块，此时会加载(返回 true)。在这之后，另一个线程也需要此模块，此时会加载，但是因为前一个线程已经加载过了，所以这里会失败(返回 false).

也就是说，autoload 不是线程安全的。(准确点，应该是 Ruby 2.0 之前的版本不是线程安全的)

使用 eager_autoload

为每个线程声明一下要加载的模块(纪录在各自的 @_autoloads)，真正需要的时候才会加载。因为每个线程都有声明，所以一个线程是否加载，并不会影响另一进程。

并且，很好的解决了 Ruby 旧版本不运行 autoload 的困境。

使用举例

```
module MyLib
  extend ActiveSupport::Autoload

  autoload :Model

  eager_autoload do
    autoload :Cache
  end
end
```

```
MyLib.eager_load!
```

其它方法

除上述常用方法外，还有：

```
autoload_at
autoload_under

autoloads
```

关于 autoload_paths 和 eager_load_paths

原因：

- 开发环境，启动速度要快（不完整也没关系）
- 生产环境，项目要完整（启动速度慢点也没关系）
- 开发环境，经常更改代码，不用重启就能运行
- 生产环境，假设不能/不会更改代码
- 旧版本使用自动加载还面临线程安全问题，在这生产环境是不能忍受的

怎么自动加载？

按约定来...

```
# 总开关，开发、生产环境不同
eager_load
```

```
# 立即加载此路径下的文件
eager_load_paths

# 自动(延迟)加载此路径下的文件
autoload_paths
```

例外：app/ 既在 autoload_paths, 又在 eager_load_paths.

什么情况下，你配置了也没用？

开发环境配置立即加载，仍然按自动加载执行。
生产环境下只配置自动加载，为了线程安全等考虑，需要再加上立即加载。

根据环境，互相影响：

在开发环境，使用 eager_load_paths 会影响 autoload_paths，等同于使用 autoload_paths；
在生产环境不影响，按照它本来的意思运行。

使用 autoload_paths 始终不影响 eager_load_paths.

注意：上述表达是不严谨的！

即使是生产环境下，大部分情况使用 autoload_paths 代替 eager_load_paths 也是可以工作的，但会存循环依赖等风险，偶尔会出问题。
所以，如果你要延迟加载，推荐一直使用 eager_load_paths.

直接对 lib/ 用 eager_load_paths ？

想法：反正在开发环境是自动加载，在生产环境是立即加载。
弊端：用的是 lib/*/.rb 模式，生产环境会把多余的 tasks、generators 也立即加载了。除非你不在乎。

require_dependency 如何使用？

initializers 和 tasks 只执行一次。

Lazy Load Hooks

允许 Rails 延迟加载部分组件，从而加快应用的启动速度。

类方法：

```
on_load
run_load_hooks

execute_hook
```

使用举例：

```
# on_load 方法触发标识及内容
initializer 'active_record.initialize_timezone' do
  ActiveSupport.on_load(:active_record) do
    self.time_zone_aware_attributes = true
    self.default_timezone = :utc
  end
end
```

```
# run_load_hooks 方法执行
ActiveSupport.run_load_hooks(:active_record, ActiveRecord::Base)
```

单独使用举例：

```
require 'active_support/lazy_load_hooks'

class Say
  def initialize(name)
    @name = name

    ActiveSupport.run_load_hooks(:instance_of_color, self)
  end
end

ActiveSupport.on_load :instance_of_color do
  puts "Hi #{@name}"
end

Say.new("kelby")
# => "Hi kelby"
```

在 Rails 框架源代码里，大量使用了它。

Concern

可以方便快捷的扩展某个类或模块，并且处理了潜在的依赖问题。

作用有二：

1) 更简洁、明了的语法。
2) 更好的处理模块之间的依赖关系，规避潜在的模块之间的循环依赖。

约定：ClassMethods 和 class_methods 自动继承、实例方法自动包含、自动使用 class_eval.

原来你需要手动写 included 或 extended 里面的代码，有实例方法、类方法的话，也要手动包含或继承；现在按照约定来即可。

以前：

```
module M
  def self.included(base)
    base.extend ClassMethods

    base.class_eval do
      # 执行某些方法
      scope :disabled, -> { where(disabled: true) }

      # 执行某些方法
      include InstanceMethods
    end
  end

  # 定义类方法
  module ClassMethods
    # ...
  end

  # 定义实例方法
  def a_instance_method
    # ...
  end

  # 如果实例方法比较多，可以单独成 module，对应上面的 include InstanceM
ethods
  module InstanceMethods
    # ...
  end
end
```

现在：

```
require 'active_support/concern'

module M
  extend ActiveSupport::Concern

  included do
    # 执行某些方法
    scope :disabled, -> { where(disabled: true) }

    # 执行某些方法
    include InstanceMethods
  end

  # 定义类方法
  module ClassMethods
    # ...
  end

  # 定义实例方法
  def a_instance_method
    # ...
  end

  # 如果实例方法比较多，可以单独成 module，对应上面的 include InstanceM
ethods
  module InstanceMethods
    # ...
  end
end
```

本模块源代码、及示例已经经过多次更改，多学语法，适应改变适应其变化。

File Update Checker

文件更新检测(文件更新后，不用重启自动编译、生效)。

Rails 默认有 3 个：

- i18n_reloader(I18n 相关)
- routes updater(Routes 相关)
- file_watcher(其它所有)

我们定制的可以直接使用 `Rails.configuration.file_watcher` 放到"其它所有"里，也可以直接调用 ActiveSupport::FileUpdateChecker.

```
# 定义。这里的 paths 表示你要监视的文件或目录(所在的路径)。
i18n_reloader = ActiveSupport::FileUpdateChecker.new(paths) do
  I18n.reload! # 监视内容有变化时，执行什么操作。
  # Rails.application.reload_routes!
end

ActionDispatch::Reloader.to_prepare do # to_prepare 的作用，可参考对应说明。
  # 调用。如果有更新，则执行操作
  i18n_reloader.execute_if_updated
end

# 调用。手动执行操作
i18n_reloader.execute
```

可用 `Rails.application.reloaders` 查看所有检测器(要先放进来，才能查看)。

```
Rails.application.reloaders << i18n_reloader
```

除上述方法外，还有：

```
updated?
```

用于询问监视的文件/目录是否有变化。

Notifications

发布、订阅功能。怎么使用？

- subscribe - 订阅
- instrument - 发布

```
# 发布消息
ActiveSupport::Notifications.instrument('render', extra: :inform
ation) do
  # 下面是真正要执行的内容
  render text: 'Foo'
end

# 订阅消息
ActiveSupport::Notifications.subscribe('render') do |name, start
, finish, id, payload|
  # 以下 4 个属性是 notification 自带的
  name    # => 类型是字符串, 代表 notification 的名字(在这里是 'rende
r')
  start   # => 类型是 Time, 代表上面开始"执行内容"的时间
  finish  # => 类型是 Time, 代表上面结束"执行内容"的时间
  id      # => 类型是 String, 唯一的 ID 表示此 notification

  # 以下属性对应着 instrument 里的 extra
  payload # => 类型是 Hash, 也就是上面传递过来的参数
end
```

> Note: 可以按正则规则进行"订阅"。

为什么使用？

Rails 内外，都有很多类似、可替换的技术。但这个方法，即能保证在同一项目内进行，又能使耦合度尽可能的降低。并且，它使用范围很广 。

缺点：配置不方便，一启动就执行，并且更改要重启。默认对性能是有影响的，subscribe 并不是异步执行。

Rails 默认有很多 Instrumentation，你可以不写 instrument 直接 subscribe 它们。或者，你自己写 instrument，然后 subscribe.

有以下特点：

- 低耦合，并且轻量化
- 操作简单
- 随处可用

实现：运用中间变量 notifier.

> Note: 注意使用场景。这里并不是严格的发布者、订阅者模型，你从它们的方法名及默认参数就应该知道。

在控制台里，可运行以下命令，查看 Rails 项目里有哪些 instrumenter：

```
ActiveSupport::Notifications.instrumenter.class
# => ActiveSupport::Notifications::Instrumenter

fanout = ActiveSupport::Notifications.instrumenter.instance_vari
able_get("@notifier")

fanout.class
# => ActiveSupport::Notifications::Fanout

fanout.instance_variable_get("@subscribers").map do |s|
  s.instance_variable_get "@pattern"
end
```

> Note: 此命令包含的 instrumenter 并不完整。比如：元编程创建的 instrumenter 就不包含。(Action Controller 有很多 instrumenter 都是元编程生成的)

所有方法:

```
instrument
instrumenter

publish

subscribe
subscribed
unsubscribe
```

Rails 默认已有的 instrumenter

action_mailer

```
deliver.action_mailer
receive.action_mailer
process.action_mailer
```

action_controller

```
write_fragment.action_controller
read_fragment.action_controller
exist_fragment?.action_controller
expire_fragment.action_controller

start_processing.action_controller
process_action.action_controller
send_file.action_controller
send_data.action_controller
redirect_to.action_controller
halted_callback.action_controller

unpermitted_parameters.action_controller

deep_munge.action_controller
```

action_view

```
render_collection.action_view
render_partial.action_view

render_template.action_view
```

active_job

```
perform_start.active_job
perform.active_job

enqueue_at.active_job
enqueue.active_job
```

active_record

```
instantiation.active_record
sql.active_record
```

active_support

```
cache_read.active_support
cache_generate.active_support
cache_fetch_hit.active_support
cache_write.active_support
cache_delete.active_support
cache_exist?.active_support

cache_delete_matched.active_support

cache_increment.active_support
cache_decrement.active_support

cache_cleanup.active_support
cache_prune.active_support
```

rails

```
deprecation.rails
```

railties

```
load_config_initializer.railties
```

参考

Active Support Instrumentation

Subscriber

只需继承于它，并定义同名方法，然后 attach_to 即可。

类方法：

```
attach_to

method_added

subscribers
```

`attach_to` 会把子类里的实例方法让 Notifications 的实例对象进行 subscribe (调用 add_event_subscriber)。

其它类方法：

```
add_event_subscriber
```

`add_event_subscriber` 核心实现。

实例方法：

```
start
finish
```

使用举例：

```
module ActiveRecord
  class StatsSubscriber < ActiveSupport::Subscriber
    def sql(event)
      Statsd.timing("sql.#{event.payload[:name]}", event.duratio
n)
    end
  end
end

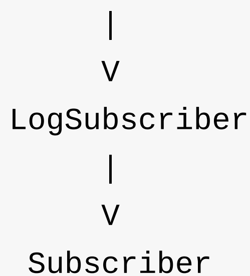
ActiveRecord::StatsSubscriber.attach_to :active_record
```

使用到了 Notifications，依赖于 Notifications 里面的"订阅"。本质还是 Notifications 的订阅，只是为了方便使用而创建出来的。

> Note: 原 Notifications 的 instrument 不变，但 subscribe 太麻烦了，所以才有 Subscriber 以及它的各个子类。

Log Subscriber

继承于 Subscriber.



```
Rails 各个模块里的 LogSubscriber
         |
         V
    LogSubscriber
         |
         V
     Subscriber
```

类方法：

```
flush_all!

log_subscribers

logger
```

实例方法：

```
finish
start

logger
```

其它实例方法：

```
info
debug
warn
error
fatal
unknown
```

和

```
color
```

使用举例：

```
module ActiveRecord
  class LogSubscriber < ActiveSupport::LogSubscriber
    def sql(event)
      "#{event.payload[:name]} (#{event.duration}) #{event.paylo
ad[:sql]}"
    end
  end
end

ActiveRecord::LogSubscriber.attach_to :active_record
```

Action Mailer Log Subscriber

日志记录，继承于 ActiveSupport::LogSubscriber，执行哪个方法时想要记录日志，只需要创建和它同名方法，然后打印日志即可。Log Subscriber 章节会讲到。

目前 Rails 侦听以下方法：

```
deliver # 发送
receive # 接收
process # 处理
```

除上述方法外，还有：

```
logger
```

Action Controller Log Subscriber

```
deep_munge

halted_callback

process_action

redirect_to

send_data
send_file

start_processing

unpermitted_parameters


# 以下几个方法使用了元编程来定义，API 里是看不到的

read_fragment
write_fragment
expire_fragment

exist_fragment?

write_page
expire_page
```

除上述方法外，还有：

```
logger
```

Action View Log Subscriber

```
from_rails_root

rails_root

render_collection & render_partial & render_template
```

除上述方法外，还有：

```
logger
```

Active Record Log Subscriber

类方法：

```
runtime
runtime=

reset_runtime
```

实例方法：

```
odd?

render_bind

sql
```

除上述方法外，还有：

```
logger
```

Rescuable

类方法：

`rescue_from(*klasses, &block)`

klasses 表示一个或多个异常类。如果有 :with 选项，则用其 value(一般是个方法)处理；否则，需要传递 block 来处理。

```
class ApplicationController < ActionController::Base
  # 一个或多个异常类，有 :with 选项
  rescue_from User::NotAuthorized, with: :deny_access # 自定义的异常处理方法
  rescue_from ActiveRecord::RecordInvalid, with: :show_errors

  # 一个或多个异常类，传递 block
  rescue_from 'MyAppError::Base' do |exception|
    render xml: exception, status: 500
  end

  protected
    def deny_access
      # ...
    end

    def show_errors(exception)
      exception.record.new_record? ? ...
    end
end
```

实现：类似'实例变量的运用'，`rescue_from` 把希望捕捉的异常类放到一个变量(rescue_handlers)里。抛出异常时，会对异常进行检查(rescue_with_handler)，如果抛出的异常恰好被包含在这个变量(rescue_handlers)，则用我们的方式进行处理。

实例方法：

```
handler_for_rescue

rescue_with_handler
```

~~Descendants Tracker~~

查看某个类的子类。功能上和 Ruby 内置库 Object Space 类似，但性能上要比它好得多。

```
class A
  extend ActiveSupport::DescendantsTracker
end

class B < A
end

class C < A
end

class D < B
end

# 输出 A 的所有子类
A.descendants
=> [B, D, C]

# 输出 A 的所有直接子类
A.direct_descendants
=> [B, C]
```

手法：重写 inherited 方法，当发生继承关系时，记录到 `@@direct_descendants` 里。

~~Dependencies~~

依赖。

Class Cache

```
[] & get

clear!

empty?

key?

safe_get

store
```

Watch Stack

```
each

new_constants

watch_namespaces

watching?
```

另：

自动加载会从 app/models, app/controllers, lib/ and other load paths.等加载，如果没有会：

```
# in active_support/dependencies.rb
def const_missing(const_name)
  from_mod = anonymous? ? guess_for_anonymous(const_name) : self
  Dependencies.load_missing_constant(from_mod, const_name)
end
```

然后：

```
#lib/active_support/dependencies.rb:477
expanded = File.expand_path(file_path)
expanded.sub!(/\.rb\z/, '')

if loading.include?(expanded)
  raise "Circular dependency detected while autoloading constant
#{qualified_name}"
end
```

此处有魔法。

Active Support eager_autoload 的类和模块

TODO

Cache 缓存的源头

类方法：

`expand_cache_key(key, namespace = nil)`

让 cache_key 过期。

使用举例：

```
expand_cache_key([:foo, :bar])              # => "foo/bar"
expand_cache_key([:foo, :bar], "namespace")  # => "namespace/foo
/bar"
```

`lookup_store(*store_option)`

指定缓存存储方式。

使用举例：

```
ActiveSupport::Cache.lookup_store(:memory_store)
# => 返回一个 ActiveSupport::Cache::MemoryStore 实例对象

ActiveSupport::Cache.lookup_store(:mem_cache_store)
# => 返回一个 ActiveSupport::Cache::MemCacheStore 实例对象
```

Store

操作的对象是下面的 entry

是 FileStore, MemCacheStore, MemoryStore, NullStore 以及 LocalStore 的基类，某些方法需要子类重写才有意义。

实例方法：

```
cleanup
clear

decrement
increment

delete
delete_matched

exist?

mute

read
write
read_multi
fetch
fetch_multi

silence!
```

`fetch` 使用举例：

```
cache.write('today', 'Monday')
cache.fetch('today')  # => "Monday"

cache.fetch('city')   # => nil
cache.fetch('city') do
  'Duckburgh'
end
cache.fetch('city')   # => "Duckburgh"
```

其它实例方法：

```
key_matcher
```

1) File Store (文件存储)

```
cleanup
clear

decrement
increment

delete_matched
```

2) Mem Cache Store (缓存存储)

```
clear

read_multi

stats
```

3) Memory Store (内存存储)

```
cleanup
clear
prune

decrement
increment

delete_matched

pruning?
```

4) ~~Null Store~~

不使用任何介质进行存储，在开发、测试环境可能有用。

~~Local Cache (本地缓存)~~

使用上述的几种介质进行存储，再快，也没有直接使用内存来得快。上述几种介质存储只要实现"本地缓存"，那么在一个 block 里首次调用变量时，会备份它到本地缓存，之后在 block 里再次调用(相同 cache key)，则直接使用本地缓起来存的，在这性能上比之前又更快了一点。

Local Cache

简单的 in-memory 缓存。

实例方法：

```
middleware

with_local_cache
```

其它实例方法：

```
set_cache_value
```

属于 middleware, rake middleware 里就能看到，并且一开始就执行了。

Local Store

简单的 memory 存储。(非线程安全)

```
clear

delete_entry
read_entry
write_entry
```

~~Entry~~

表示某一条 cache 数据，包含了值和过期时间等信息。

Callback 方法解释及使用

这里是 Rails 所有回调的源头。从表面看，它提供以下方法。

类方法：

```
define_callbacks

set_callback
skip_callback

reset_callbacks
```

实例方法：

```
run_callbacks
```

已有回调类型：

```
CALLBACK_FILTER_TYPES = [:before, :after, :around]
```

使用过程中，3 个必不可少的方法及其解释如下：

方法	解释
define_callbacks	定义一条"回调链"。
set_callback	把某种类型的"回调"放到"回调链"里。 需要 3 个参数：回调链的名字、回调的类型(不传递的话，自动使用 :before)、回调的名字。
run_callbacks	在执行代码的前后，执行指定"回调链"相关的"回调"。 第一个参数是"回调链"的名字，第二个参数是个 block，里面放真正要执行的代码。

使用举例：

```
class Report
  include ActiveSupport::Callbacks

  # 定义一条回调链，名字是 print
  define_callbacks :print

  # 把类型为 before，名字为 before_print 的回调，放到 print 回调链里
  set_callback :print, :before, :before_print
  # 把类型为 after，名字为 after_print 的回调，放到 print 回调链里
  set_callback :print, :after, :after_print

  def print_me
    # 在执行"真正要执行的代码"的前后，执行 print 回调链里的回调
    run_callbacks :print do
      # 真正要执行的代码
      p 'print me'
    end
  end

  # 以下两方法已经放入 print 回调链，所以会被调用。

  def before_print
    p 'before print'
  end

  def after_print
    p 'after print'
  end
end

Report.new.print_me
# => "before print"
# => "print me"
# => "after print"
```

Callbacks 底层简要分析

ActiveSupport::Callbacks 本身可分为几部分。

Callback Chain 回调链

`define_callbacks` 主要作用就是定义一条"回调链"，每一条链都是 Callback Chain 的实例对象。

Callback Chain 实例对象，大致如下：

```
#<ActiveSupport::Callbacks::CallbackChain:0x007fc6ebf70648
  @callbacks=nil,
  @chain=[],
  @config={:scope=>[:kind]},
  @mutex=#<Mutex:0x007fc6ebf70580>,
  @name=:checkout>
```

其中，最重要的信息有：

- `@chain` 此回调链所包含的"所有回调"。
- `@name` 此回调链的"名字"。

Rails 没有提供查看所有 callback chain 信息的接口，只能间接查看。

比如：

```
# 所有"回调链"
callback_chains = ObjectSpace.each_object(ActiveSupport::Callbac
ks::CallbackChain)
# 不为空的"回调链"
chains = callback_chains.select{|cc| cc.instance_variable_get("@
chain").present? }

# 所有"回调链"的名字
callback_chains_uniq_name = chains.map(&:name).uniq
```

但对于某个类而言，则很方便。如：

```
callback_chains = ClassName.singleton_methods.select do |method|
  method.to_s =~ /^_[^_].+_callbacks$/
end

# Rails model 默认有 callback chains：
:_save_callbacks
:_create_callbacks
:_update_callbacks
:_validate_callbacks
:_validation_callbacks
:_initialize_callbacks
:_find_callbacks
:_touch_callbacks
:_destroy_callbacks
:_commit_callbacks
:_rollback_callbacks

# 查看某 callback chain 详情。如：
User._save_callbacks
```

Callback 回调

上面提到每一条回调链的 @chain 里包含了它"所有的回调"，这里的每一个"回调"就是一个 Callback 实例对象。

Callback 实例对象，大致如下：

```
#<ActiveSupport::Callbacks::Callback:0x007fc6f2162248
  @chain_config={:scope=>[:kind]},
  @filter=#<Proc:0x007fc6f2162388@/Users/.../lib/rails/applicati
on/finisher.rb:100>,
  @if=[],
  @key=70246220894660,
  @kind=:before,
  @name=:prepare,
  @unless=[]
```

其中，最重要的信息有：

- `@if` 或 `@unless` 此回调起作用的"前提条件"。
- `@kind` 此回调的"回调类型"。(对于 Rails 而言，可以选择 :before, :after, :around 其中之一)
- `@name` 此回调所加入的"回调链的名字"。
- `@key` 此回调的"名字"。(如果没有名字则用 object_id 代替)
- `@filter` 此回调"真正要执行的代码"。(一般只显示名字，也就是说和 @key 一样；如果没有名字，则显示定义它时的文件及行号)

"真正要执行的代码"，可以是以下类型:

```
Symbols:: 方法名。
Strings:: 可求值的字符串。
Procs::   proc 对象。
Objects:: 普通的实例对象，但必需有 before_x, around_x, after_x 等"回调"方法。
          因为，之后会以拼接字符串的形式，找出对应的回调方法，然后 send 调用。

其实，还有一种类型 Conditionals 但被直接跳过了，所以不算在内。
```

这些不同类型的"真正要执行的代码"，之后都会被 Callback 的 `make_lambda` 方法转换成 lambda 对象，再然后处理过程类似。

其它

当 **@filter** 是一个实例对象，并且恰好有和**"**回调类型**"**相同的方法

上面也提到"回调类型"，Rails 默认有 :before、:after 和 :around.
上面提到了回调里"真正要执行的代码"可以是一个实例对象，当触发回调时，会执行它的同名"回调"方法。

如果 @filter 恰好是一个实例对象，而这个实例对象又恰好有 :before、:after 或 :around，则此时会出现和想像中不一样的结果。

比如：

```
require 'active_support'

class Audit
  def before(caller)
    puts 'Audit: before is called'
  end

  def before_save(caller)
    puts 'Audit: before_save is called'
  end
end

class Account
  include ActiveSupport::Callbacks

  define_callbacks :save
  set_callback :save, :before, Audit.new

  def save
    run_callbacks :save do
      puts 'save in main'
    end
  end
end

Account.new.save
```

此时，我们可以使用 `define_callbacks` 的 scope 参数进行解决。也就是：

```
define_callbacks :save, scope: [:kind, :name]
```

Rails 里 Callback 相关的模块及继承关系

```
ActiveRecord::Callbacks
           |
           V
ActiveModel::Callbacks
           |
           V
ActiveSupport::Callbacks
```

```
ActiveModel::Validations::Callbacks
ActiveJob::Callbacks
ActionDispatch::Callbacks
AbstractController::Callbacks
         |
         V
ActiveSupport::Callbacks
```

Filters 顺序

一条"回调链"上可以有多个"回调"，它们彼此之间不是独立的，有先后顺序。即：

- Before，After，Around

但这些"回调"也有终结的时候。即：

- End

定义、运行回调

`define_callbacks` 定义的时候会：

```
define_callbacks
       |
       V
set_callbacks name, CallbackChain.new(name, options)  # 这里的 na
me 表示"回调链"的名字。

def set_callbacks(name, callbacks) # 这里的 callbacks 是"回调链"的
实例对象。
  send "_#{name}_callbacks=", callbacks
end

define_callbacks
       |
       V
def _run_#{name}_callbacks(&block)
  _run_callbacks(_#{name}_callbacks, &block)
end
```

`run_callbacks` 运行的时候会：

```
run_callbacks
      |
      V
_run_#{kind}_callbacks # 这里的 kind 表示"某种类型的"回调链
      |
      V
_run_callbacks(_#{name}_callbacks, &block) # 这里的 name 表示"回调
链"
      |
      V
runner = callbacks.compile # 这里的 callbacks 表示"回调链"；
                           # compile 会创建 Callback 实例对象并做后
续处理。
e = Filters::Environment.new(self, false, nil, block)
runner.call(e).value
```

Configurable

实例方法

`config` 用 @_config 实例变量来保存配置信息。

使用举例：

```
require 'active_support/configurable'

class User
  include ActiveSupport::Configurable
end

user = User.new

# 写 config 对象
user.config.allowed_access = true
user.config.level = 1

# 读 config 对象
user.config.allowed_access # => true
user.config.level          # => 1
```

类方法

`config_accessor` 以声明的形式，同时定义类方法和实例方法。 实例对象的值默认继承于类对象，修改某个实例对象的值不影响类对象和其它实例对象的值。

使用举例：

```
class User
  include ActiveSupport::Configurable

  # 直接使用
  config_accessor :allowed_access
end

# 相关类方法
User.allowed_access # => nil
User.allowed_access = false
User.allowed_access # => false

user = User.new

# 相关实例方法
user.allowed_access # => false
user.allowed_access = true
user.allowed_access # => true

User.allowed_access # => false
```

再次举例：

```
class User
  include ActiveSupport::Configurable

  # 带 instance_reader、instance_writer 参数
  config_accessor :allowed_access, instance_reader: false, insta
nce_writer: false

  # 带 instance_accessor 参数
  config_accessor :allowed_access, instance_accessor: false

  # 带 block
  config_accessor :hair_colors do
    [:brown, :black, :blonde, :red]
  end
end
```

除此之外，还有类方法：

```
config # Configuration 的实例对象。实例方法 config 和 类方法 config_accessor 调用到它。

configure # 直接封装 config
```

> Note: 它和 railties 目录下的 Configurable 和 Configuration 都没有关系。目前只发现有 AbstractController::Base 引用到它(其子类由于继承关系，也可以使用)。

~~Configuration~~

```
# 类方法
compile_methods!

# 实例方法
compile_methods!
```

另外，它继承于 ActiveSupport::InheritableOptions

> Note: 不对 module Configurable 之外提供接口，只有这里使用到它。

两部分: Ordered Options 和 Ordered Hash.

Ordered Options

以方法的形式调用 Hash 的 key，读、写其 value.

普通 Hash：

```
h = {}

h[:boy]  = 'John'
h[:girl] = 'Mary'

h[:boy]  # => 'John'
h[:girl] # => 'Mary'
```

使用 Ordered Options 后：

```
h = ActiveSupport::OrderedOptions.new

h.boy  = 'John'
h.girl = 'Mary'

h.boy  # => 'John'
h.girl # => 'Mary'

# 原来的调用方式仍然有效

h[:boy]  = 'John'
h[:girl] = 'Mary'

h[:boy]  # => 'John'
h[:girl] # => 'Mary'
```

Ordered Hash

默认，Ruby 的 Hash 已经按照加入时的顺序进行排序，但这一点并不能得到保证，因为可能有"猴子补丁"的作用。

引入此模块后可，至少可当做 namespace，解决这一问题。

```
oh = ActiveSupport::OrderedHash.new
oh[:a] = 1
oh[:b] = 2

# 确保 key 的顺序和加入时一样
oh.keys
# => [:a, :b]
```

Inflector

String 扩展里的 inflections 全部是直接封装这里的方法而来。

methods

```
pluralize
singularize

camelize
underscore

humanize
titleize

tableize
classify

dasherize
demodulize
deconstantize
foreign_key
constantize
safe_constantize
ordinal
ordinalize

inflections
parameterize
transliterate
```

transliterate

```
transliterate
parameterize
```

inflections

```
acronym

clear

human

irregular

plural
singular

uncountable
```

Key Generator 和 Caching Key Generator

相同点，都提供方法：

```
generate_key
```

不同点：

Key Generator 生成的是初级密钥。

```
key_generator = ActiveSupport::KeyGenerator.new 'secret'
key_generator.generate_key 'salt'
```

Caching Key Generator 再次封装 Key Generator 和 Thread Safe，生成的是高级密钥。

```
key_generator = ActiveSupport::KeyGenerator.new 'secret'

# 参数是 KeyGenerator 实例对象
caching_key_generator = ActiveSupport::CachingKeyGenerator.new(k
ey_generator)
caching_key_generator.generate_key 'salt'
```

> Note: 注意和【Message Encryptor 和 Message Verifier】章节的联系。

两部分: Message Verifier 和 Message Encryptor.

Message Verifier

生成加密的文本，然后用于校验。这里的加密仅意味着"加签名、防篡改"，过程是可逆的，请注意使用场景。 使用场景举例，生成"记住我"的 token，或生成"取消订阅"的链接。

```
verifier = ActiveSupport::MessageVerifier.new('your-secret')
message = "String that is prevented from tampering."

# 签名(加密)
signed_message = verifier.generate(message)

# 验证(解密)
verified = verifier.verify(signed_message)

# 比较
verified == message
# => true
```

主要是 `generate(value)` 和 `verify(signed_message)`，原理很简单，这里不多赘述。

另，验证加密信息是否被篡改：

```
verifier = ActiveSupport::MessageVerifier.new 's3Krit'
signed_message = verifier.generate 'a private message'
verifier.valid_message?(signed_message) # => true

tampered_message = signed_message.chop # 篡改"已经签名"的信息
verifier.valid_message?(tampered_message) # => false
```

其它方法：

```
verified

valid_message?
```

Message Encryptor

和 Message Verifier 本质上没有区别。但使用上会更严格一点，会多一些步骤(相应地，也更安全了一点)，并且它也有调用到 Message Verifier.

还有一个优点：Message Encryptor 生成的字符串会比较短，比 Message Verifier 生成的更适合直接放到 url 里。

使用举例:

```
salt = SecureRandom.random_bytes(64)
secret_key_base = '-- secret key base --'

key_generator = ActiveSupport::KeyGenerator.new(secret_key_base)
# 生成密钥
key = key_generator.generate_key(salt)

encryptor1 = ActiveSupport::MessageEncryptor.new(key)
encryptor2 = ActiveSupport::MessageEncryptor.new(key)

message = "Encrypted string."

# 加密
encrypted_message = encryptor1.encrypt_and_sign(message)

# 解密
decrypted = encryptor2.decrypt_and_verify(encrypted_message)

# 比较
decrypted == message
# => true
```

主要是 `encrypt_and_sign(value)` 和 `decrypt_and_verify(value)` ，同样这里不多赘述。

Message Encryptor、Message Verifier 和 Key Generator 这三者使用类似，创建时可传递字符串做为参数，然后进行加密，生成的也是字符串。

实例参考

```
salt  = SecureRandom.random_bytes(64) # 保存进数据库
secret_key_base = '-- secret base --' # 保存到配置文件

string = 'string'                     # 运行于代码
secret = Digest::MD5.hexdigest("#{secret_key_base}-#{string}")
key    = ActiveSupport::KeyGenerator.new(secret).generate_key(sa
lt)
crypt1 = ActiveSupport::MessageEncryptor.new(key)

string = 'string'                     # 运行于代码
secret = Digest::MD5.hexdigest("#{secret_key_base}-#{string}")
key    = ActiveSupport::KeyGenerator.new(secret).generate_key(sa
lt)
crypt2 = ActiveSupport::MessageEncryptor.new(key)

source_data    = 'my secret data'                # 加密前数据
encrypted_data = crypt1.encrypt_and_sign(source_data) # 加密后数据
decode_data    = crypt2.decrypt_and_verify(encrypted_data)
source_data == decode_data
```

> Note: 注意和【Key Generator 和 Caching Key Generator】章节的联系。

String Inquirer - Rails.env.production?

原来的代码：

```
Rails.env == 'production'
```

语法糖：

```
Rails.env.production?
```

仅作用于 `Rails.env` 对象，方法是动态生成的，所以 API 里查询不到。

手法：把方法最后的问号去掉，看是否和当前环境一样。

Array Inquirer

元素后面加 `?` ，可直接查询是否在数组内：

```
variants = ActiveSupport::ArrayInquirer.new([:phone, :tablet])

variants.phone?    # => true
variants.tablet?   # => true
variants.desktop?  # => false
```

使用 `any?` 查询元素是否在数组内：

```
variants = ActiveSupport::ArrayInquirer.new([:phone, :tablet])

variants.any?                          # => true
variants.any?(:phone, :tablet)         # => true
variants.any?('phone', 'desktop')      # => true
variants.any?(:desktop, :watch)        # => false
```

Tagged Logging

封装了 Ruby 标准的 Logger，让我们能够给 logger 实例对象打"标签"。

```
logger = ActiveSupport::TaggedLogging.new(Logger.new(STDOUT))

logger.tagged('BCX') { logger.info 'Stuff' }
# => Logs "[BCX] Stuff"
```

"标签"还可以层层叠加：

```
logger.tagged('BCX', "Jason") { logger.info 'Stuff' }
# Logs "[BCX] [Jason] Stuff"

logger.tagged('BCX') { logger.tagged('Jason') { logger.info 'Stu
ff' } }
# Logs "[BCX] [Jason] Stuff"
```

Rails.logger 默认已经使用。

实现：封装了 Logger 的实例对象，覆盖了它自带的 Formatter 对象，并添加了 tagged 方法。

Gzip 与 JSON

Gzip

封装了标准库 zlib，提供 gzip 压缩/解压缩字符串功能。

```
# 压缩
gzip = ActiveSupport::Gzip.compress('compress me!')
# => "\x1F\x8B\b\x00o\x8D\...\x83\f\x00\x00\x00"

# 解压缩
ActiveSupport::Gzip.decompress(gzip)
# => "compress me!"
```

JSON

提供 json 格式的编码/解码。

`encode(value, options = nil)` 将对象转换成 JSON 格式

```
# 加密
# 是模块方法
ActiveSupport::JSON.encode({ team: 'rails', players: '36' })
# => "{\"team\":\"rails\",\"players\":\"36\"}"
```

`decode(json, options = {})` 将 JSON 字符串转换成 Hash

```
# 解密
# 是类方法
ActiveSupport::JSON.decode("{\"team\":\"rails\",\"players\":\"36
\"}")
=> {"team" => "rails", "players" => "36"}
```

Backtrace Cleaner

报错或程序运行反馈信息过多、过杂，找不到关键点？你可以使用 Backtrace Cleaner 过滤无用信息。

单独使用

三步即可：

```
# 1) 创建对象
bc = BacktraceCleaner.new

# 2) 添加规则。可指定'替换'某字符串信息：
bc.add_filter   { |line| line.gsub(Rails.root, '') }
# 2) 添加规则。可指定'删除'某 gem 信息：
bc.add_silencer { |line| line =~ /mongrel|rubygems/ }

# 3) 执行过滤。从 exception.backtrace 里过滤掉符合规则的字符串
bc.clean(exception.backtrace) #
```

除以上例子使用到的方法外，还有：

方法	参数
filter & clean	执行过滤。参数是要处理的信息
remove_filters!	移除之前的'替换'规则
remove_silencers!	移除之前的'删除'规则

结合 Rails 使用

Rails 启动时就已使用 backtrace_cleaner, 并且抛异常时会对异常消息进行过滤。

也就是说，Rails 已经帮我们做了第 1 和第 3 步，我们只要做第 2 步"添加规则"即可。 可以在以下配置文件，设置过滤条件：

```
# config/initializers/backtrace_silencers.rb
Rails.backtrace_cleaner.add_silencer { |line| line =~ /my_noisy_
library/ }
```

通过以下方式，可以查看 backtrace_cleaner 的 filters 和 silencers 信息：

```
Rails.backtrace_cleaner
=> #<Rails::BacktraceCleaner:0x007ff859bed328
 @filters=
  [#<Proc:0x007ff859bed238@/Users/.../lib/rails/backtrace_cleane
r.rb:10>,
   #<Proc:0x007ff859bed210@/Users/.../lib/rails/backtrace_cleane
r.rb:11>,
   #<Proc:0x007ff859bed1c0@/Users/.../lib/rails/backtrace_cleane
r.rb:12>,
   #<Proc:0x007ff859bec450@/Users/.../lib/rails/backtrace_cleane
r.rb:26>],
 @silencers=
  [#<Proc:0x007ff859bec428@/Users/.../lib/rails/backtrace_cleane
r.rb:15>]>
```

附(Rails 里的 backtrace_cleaner 的定义与调用)：

定义：

```
# railties/lib/rails/backtrace_cleaner.rb
module Rails
  class BacktraceCleaner < ActiveSupport::BacktraceCleaner
    # ...
  end
end

# railties/lib/rails.rb
def backtrace_cleaner
  @backtrace_cleaner ||= begin
  require 'rails/backtrace_cleaner'
  Rails::BacktraceCleaner.new
  end
end

# railties/lib/application.rb
def env_config
  # ...
  "action_dispatch.backtrace_cleaner" => Rails.backtrace_cleaner
  # ...
end
```

调用：

```
# action_dispatch/middleware/exception_wrapper.rb
def backtrace_cleaner
  @backtrace_cleaner ||= @env['action_dispatch.backtrace_cleaner'
]
end

def clean_backtrace(*args)
  if backtrace_cleaner
    backtrace_cleaner.clean(backtrace, *args)
  else
    backtrace
  end
end
```

Number Helper

```
number_to_currency
number_to_delimited
number_to_human
number_to_human_size
number_to_percentage
number_to_phone
number_to_rounded
```

```
autoload :NumberToRoundedConverter
autoload :NumberToDelimitedConverter
autoload :NumberToHumanConverter
autoload :NumberToHumanSizeConverter
autoload :NumberToPhoneConverter
autoload :NumberToCurrencyConverter
autoload :NumberToPercentageConverter
                |
                V
autoload :NumberConverter
```

Benchmarkable

耗时统计

执行某程序，用了多少时间？

```
benchmark
```

使用举例：

```
<% benchmark 'Process data files', level: :info, silence: true do
 %>
  <%= expensive_and_chatty_files_operation %>
<% end %>
```

可先用 gem 'newrelic' 等大致定位，然后使用 benchmark 实现精确定位。

链接 标准库 Benchmark

Xml Mini

Xml Mini 属于接口， `to_xml` 底层处理之一，以及允许你切换使用不同的库进行解析 HTML/XML.

默认用的是标准库 REXML，不过你可以使用性能更快的 LibXML 或 Nokogiri. (性能上快得多，不过有极少数不规范的网站可能会解析失败)

```
ActiveSupport::XmlMini.backend
# => ActiveSupport::XmlMini_REXML
```

类方法(对外接口)：

```
backend
backend=

rename_key

to_tag

with_backend
```

此外，还有类方法：

```
delegate :parse, :to => :backend
```

`parse` 是主要对外接口。前台保持它不变即可，后台可随意更改解析器。

可选择的库：

```
XmlMini_JDOM
XmlMini_LibXML
XmlMini_LibXMLSAX
XmlMini_Nokogiri
XmlMini_NokogiriSAX
# 这是默认的
XmlMini_REXML
```

使用示例：

```
gem 'libxml-ruby', '=0.9.7'
XmlMini.backend = 'LibXML'
```

另，Rails 里的数组和哈希的 `to_xml` 也用到了它进行转换处理。

Multibyte

使处理各种字符成为可能。

类方法：

```
proxy_class
proxy_class=
```

```
ActiveSupport::Multibyte.proxy_class = CharsForUTF32
```

Chars

Rails 里很多字符串，都会先转换成它的实例对象，然后再处理。

实例方法：

```
capitalize

compose

decompose

downcase

grapheme_length

limit

normalize

respond_to_missing?

reverse

slice!

split

swapcase

tidy_bytes

titleize & titlecase

upcase
```

普通字符串通过 `mb_chars` 方法，可以得到 Chars 的实例对象，然后可以调用这里的实例方法。

```
require 'active_support/multibyte'

class String
  def mb_chars
    ActiveSupport::Multibyte.proxy_class.new(self)
  end
end
```

```
'The Perfect String  '.mb_chars.downcase.strip.normalize # => "t
he perfect string"
```

> Note: Chars 还封装使用了下面的 Unicode.

Unicode

类方法：

```
compose

decompose

downcase

in_char_class?

normalize

pack_graphemes

reorder_characters

swapcase

tidy_bytes

unpack_graphemes

upcase
```

~~Deprecation~~

指定 Rails 里准备过期的方法、实例变量、对象和常量。

```
include Singleton
include InstanceDelegator
include Behavior
include Reporting
include MethodWrapper
```

具体不做介绍。

其它

- ~~Proxy Object~~
- ~~Option Merger~~

Active Support 核心扩展

TODO

Array

Access

```
from
to

without

second
third
fourth
fifth
forty_two
```

Conversions

```
to_sentence

to_formatted_s & to_default_s & to_s

to_xml
```

Extract_options

```
extract_options!
```

Grouping

```
in_groups
in_groups_of

split
```

prepend and append

```
append & <<
prepend & unshift
```

Wrap

```
wrap
```

Array Inquirer

数组可以调用期望所包含元素结尾加 `?` 以询问是否包含此元素。

或使用 `any?` 传递元素名字询问。

其它

```
inquiry

deep_dup

to_param
to_query
```

Benchmark

```
ms
```

使用举例：

```
Benchmark.realtime { User.all }
# => 8.0e-05

# 和 realtime 一样, 但单位是"毫秒"
Benchmark.ms { User.all }
# => 0.074
```

Big Decimal

```
duplicable?
```

Class

```
class_attribute

subclasses
```

Date

```
<=> & compare_without_coercion & compare_with_coercion

acts_like_date?

blank?

advance

ago
since & in

at_beginning_of_day & beginning_of_day & midnight & at_midnight

at_end_of_day & end_of_day

midday & noon & at_midday & at_noon & middle_of_day

at_middle_of_day

beginning_of_week
beginning_of_week=
change
current

find_beginning_of_week!

inspect
default_inspect
readable_inspect

to_default_s
to_formatted_s
to_s

to_time
tomorrow
```

```
xmlschema
yesterday

plus_with_duration & + & plus_with_duration

minus_without_duration & - & minus_with_duration
```

Time

```
acts_like_time?
```

```
-
minus_with_coercion
minus_without_coercion
minus_without_duration

<=>
compare_with_coercion
compare_without_coercion

===,

acts_like_time?

advance, ago, all_day,

at_beginning_of_day
beginning_of_day
at_midnight
midnight

at_beginning_of_hour & beginning_of_hour

at_beginning_of_minute & beginning_of_minute

at_end_of_day & end_of_day

at_end_of_hour
end_of_hour

at_end_of_minute
end_of_minute

at_midday
```

```
midday

at_middle_of_day
middle_of_day
at_noon
noon

at_with_coercion

change

current

days_in_month

eql?
eql_with_coercion
eql_without_coercion

find_zone, find_zone!, formatted_offset

in
since

seconds_since_midnight, seconds_until_end_of_day

days_in_year
```

```
to_default_s & to_formatted_s & to_s

formatted_offset
```

```
zone
zone=

use_zone

find_zone
find_zone!
```

Date Time

```
<=>

acts_like_date?
acts_like_time?

at_beginning_of_day
beginning_of_day

at_beginning_of_hour
beginning_of_hour

at_beginning_of_minute
beginning_of_minute

at_end_of_day
end_of_day

at_end_of_hour
end_of_hour

at_end_of_minute
end_of_minute

at_midday
middle_of_day
noon
at_middle_of_day

advance, ago,
at_midnight, at_noon
change, civil_from_format, current
formatted_offset

in
since
```

```
inspect
default_inspect

midday, midnight

readable_inspect
seconds_since_midnight, seconds_until_end_of_day

to_f
to_i

to_s
to_default_s
to_formatted_s

nsec
usec

utc
getutc

utc?
utc_offset
```

```
on_weekend?

next_weekday
prev_weekday
```

Duration

和 Date、Time、DateTime、TimeWithZone 等时间相关类的关系比较深。

```
ago & until

from_now & since
```

使用举例：

```
1.month.ago # 等价于 Time.now.advance(months: -1)
```

Time With Zone

类似 Ruby 内置的 Time，但 Ruby 内置的 Time 所创建的实例对象局限于 UTC 和系统的 ENV['TZ'] 时区。这里的实例没有此局限，你可以用你想用的时区。

你不能直接使用 new 创建 TimeWithZone 实例对象，但可以用 local, parse, at 和 now 创建 `TimeZone` 实例对象，或者 in_time_zone 创建 `Time` 和 `DateTime` 实例对象。

```
Time.zone = 'Eastern Time (US & Canada)'
# => 'Eastern Time (US & Canada)'

Time.zone.local(2007, 2, 10, 15, 30, 45)
# => Sat, 10 Feb 2007 15:30:45 EST -05:00

Time.zone.parse('2007-02-10 15:30:45')
# => Sat, 10 Feb 2007 15:30:45 EST -05:00

Time.zone.at(1170361845)
# => Sat, 10 Feb 2007 15:30:45 EST -05:00

Time.zone.now
# => Sun, 18 May 2008 13:07:55 EDT -04:00

Time.utc(2007, 2, 10, 20, 30, 45).in_time_zone
# => Sat, 10 Feb 2007 15:30:45 EST -05:00
```

查询 Time 和 TimeZone 的 API 可以对这些方法有更多的了解。

TimeWithZone 创建的实例对象和 Ruby 内置的 Time 创建的实例对象完全兼容，也就是说它们是等价关系。

```
t = Time.zone.now                     # => Sun, 18 May 2008 13:27
:25 EDT -04:00
t.hour                                # => 13
t.dst?                                # => true
t.utc_offset                          # => -14400
t.zone                                # => "EDT"
t.to_s(:rfc822)                       # => "Sun, 18 May 2008 13:2
7:25 -0400"
t + 1.day                             # => Mon, 19 May 2008 13:27
:25 EDT -04:00
t.beginning_of_year                   # => Tue, 01 Jan 2008 00:00
:00 EST -05:00
t > Time.utc(1999)                    # => true

# 完全兼容，它们是等价关系。
t.is_a?(Time)                         # => true
t.is_a?(ActiveSupport::TimeWithZone) # => true
```

相关：Date, Time, DateTime, DateAndTime 以及 TimeZone.

实例方法：

```
+, -, <=>

acts_like_time?, advance, ago, as_json

between?, blank?

comparable_time

dst?

eql?

formatted_offset, freeze, future?

getgm, getlocal, getutc, gmt?, gmt_offset, gmtime, gmtoff

hash, httpdate
```

```
in_time_zone, inspect, is_a?, isdst, iso8601

kind_of?

localtime

marshal_dump, marshal_load, method_missing

name

past?, period

respond_to?, respond_to_missing?, rfc2822, rfc822

since, strftime

time

to_a, to_datetime, to_f, to_formatted_s, to_i, to_r, to_s, to_ti
me

today?, tv_sec

utc, utc?, utc_offset

xmlschema

zone
```

Time Zone

类方法：

```
[]

all

create & new

find_tzinfo

seconds_to_utc_offset

us_zones

zones_map
```

实例方法：

```
<=>
=~

at

formatted_offset

local
local_to_utc

now

parse

period_for_local
period_for_utc

to_s

today
tomorrow
next_day
yesterday
prev_day

utc_offset

utc_to_local
```

Enumerable

```
exclude?
many?

index_by
sum
```

File

```
atomic_write
```

Hash

实例方法：

```
assert_valid_keys

compact
compact!

deep_dup
deep_merge
deep_merge!

deep_stringify_keys
deep_stringify_keys!

deep_symbolize_keys
deep_symbolize_keys!

deep_transform_keys
deep_transform_keys!

except
except!

extract!

extractable_options?

nested_under_indifferent_access
with_indifferent_access

reverse_merge
reverse_merge! & reverse_update

slice
slice!
```

```
stringify_keys
stringify_keys!

symbolize_keys & to_options
symbolize_keys! & to_options!

to_param
to_query

to_xml

transform_keys
transform_keys!
```

类方法：

```
from_trusted_xml
from_xml
```

Hash With Indifferent Access

处理 Hash 时，使得 :foo 和 "foo" 所代表的意思是一样的。

```
rgb = ActiveSupport::HashWithIndifferentAccess.new

rgb[:black] = '#000000'
rgb[:black]  # => '#000000'
rgb['black'] # => '#000000'

rgb['white'] = '#FFFFFF'
rgb[:white]  # => '#FFFFFF'
rgb['white'] # => '#FFFFFF'
```

```
[]

[]=
regular_writer
store

convert_key
convert_value

deep_stringify_keys
deep_stringify_keys!

deep_symbolize_keys

default

delete

dup

extractable_options?

fetch
```

```
has_key?
key?
include?
member?

merge
update
regular_update
merge!

nested_under_indifferent_access
new
new_from_hash_copying_default
reject
replace

reverse_merge
reverse_merge!

select

stringify_keys
stringify_keys!

symbolize_keys

to_hash
to_options!

values_at
with_indifferent_access
```

Integer

```
month & months
year & years

multiple_of?

ordinal
ordinalize
```

Numeric

```
byte & bytes

day & days

duplicable?

exabyte & exabytes

fortnight & fortnights

from_now

gigabyte & gigabytes

hour & hours

html_safe?

in_milliseconds

kilobyte & kilobytes

megabyte & megabytes

minute & minutes

petabyte & petabytes

second & seconds

terabyte & terabytes

to_formatted_s

week & weeks
```

Kernel

```
breakpoint & debugger

capture & silence

class_eval

concern

enable_warnings

quietly

silence_stream
silence_warnings

suppress

with_warnings
```

Object

```
acts_like?

blank?

create_fixtures

deep_dup

duplicable?

html_safe?

in?

instance_values

instance_variable_names

presence
presence_in

present?

to_json_with_active_support_encoder

to_param
to_query

try
try!

unescape

with_options
```

Module

```
alias_attribute
alias_method_chain

anonymous?,

attr_internal
attr_internal_accessor

attr_internal_reader
attr_internal_writer

cattr_accessor
mattr_accessor

cattr_reader
mattr_reader

cattr_writer
mattr_writer

delegate

deprecate

foo

parent
parents
parent_name

qualified_const_defined?
qualified_const_get
qualified_const_set

redefine_method

remove_possible_method
```

alias_method_chain

不修改原有接口和调用代码，给接口添加额外的行为。

举例：

```
# 代码一
1  class Klass
2    def salute
3      puts "Aloha!"
4      # do some other things...
5    end
6  end
7
8  Klass.new.salute
9  # => Aloha!
10 # do some other things...
```

张三编写了以上代码，并提供接口给大家使用。李四用着不爽，因为没有日志记录。他想到了一个方法：

```
puts "Before do salute ..."
Klass.new.salute # => Aloha!
puts "End do salute ..."
```

李四觉得这么做非常好，于是大力推荐别人也这么做，其中就包括王五。但王五觉得李四疯了，因为这意味着要修改已有代码，并且很丑陋。

李四想了想，于是这么做...

```
# 代码二
1  class Klass
2    def salute_with_log
3      puts "Before do salute ..."
4      salute_without_log
5      puts "... End do salute"
6    end
7
8    alias_method :salute_without_log, :salute
9    alias_method :salute, :salute_with_log
10 end
```

它利用"猴子补丁"并且两次使用 `alias_method` 以达到效果，检验一下：

```
Klass.new.salute     # -> 调用代码一，第 2 行；
                     # 但代码二，第 9 行更改了调用；
                     # 实际运行的是代码二，第 2 行。

# 输出：
Before do salute ... # -> 对应代码二，第 3 行

# 接下来运行代码二，第 4 行；但代码二，第 8 行更改了调用；实际运行的是代码一，第 2 行

# 输出
Aloha!

# 接下来运行代码二，第 5 行

# 输出
... End do salute
```

完美，既没有修改张三的接口，也没有修改王五的调用代码。
只要张三提供的接口不变，王五的调用代码就不用修改，而自己的代码也能顺利运行。

但上面隐藏着一个很大的问题：代码二，第 8、9 行顺序不能颠倒，一不小心的话，将进入死循环里去...

```
Calling method...
Calling method...
Calling method...
SystemStackError: stack level too deep
```

使用 `alias_method_chain` 可以防患于未然，不必再担忧：

```ruby
class Klass
  def salute
    puts "Aloha!"
  end
end

Klass.new.salute
# => Aloha!

class Klass
  def salute_with_log
    puts "Begin do salute ..."
    salute_without_log
    puts "... End do salute"
  end
  alias_method_chain :salute, :log
end

Klass.new.salute
# =>
# Begin do salute ...
# Aloha!
# ... End do salute
```

> Rails 5 里 `alias_method_chain` 已被移除，因为 Ruby 自带 `prepend` 可以使用。

Marshal

```
load_with_autoloading
```

Range

```
overlaps?
include_with_range?

to_s
to_default_s
to_formatted_s
```

Regexp

```
multiline?
```

Secure Random

```
uuid_v3
uuid_v5

uuid_from_hash
```

String

```
acts_like_string?

at

blank?

camelcase

camelize

classify

constantize

dasherize

deconstantize

demodulize

exclude?

first

foreign_key

from

html_safe

humanize

in_time_zone

indent
indent!
```

```
inquiry

is_utf8?

last

mb_chars

parameterize

pluralize

remove
remove!

safe_constantize

singularize

squish
squish!

strip_heredoc

tableize

titlecase

titleize

to
to_date
to_datetime
to_time

truncate
truncate_words

underscore
```

URI

```
parser
```

Load Error

```
is_missing?

path
```

Name Error

```
missing_name
missing_name?
```

Logger

扩展标准库 ::Logger

Logger Silence

提供 Logger 的实例方法：

```
silence
```

使用举例：

```
Rails.logger.silence do
  # ... 这里面，"程序所输出的日志"会被清除掉。
end
```

可以通过以下设置，不用此特性：

```
ActiveSupport::Logger.silencer = false
```

此时，silence 所包含的 block 里的"程序所输出的日志"不会被清除。

链接

标准库 Logger

Active Support 其它类和模块

比如：

~~Per Thread Registry~~

Security Utils

```
secure_compare
```

`a_string == b_string` 当发现两字符串有字符不一致时，就会立即返回比较结果。

在这样的机制下, 比较时间与字符串匹配度是有正比关系的, 字符串匹配度越高，比较时间越长。因此，便有可能通过对比时间差的方式逐一猜测破解(计时攻击)。

使用此方法，即使发现两字符串有不同，也不会立即返回结果，而是继续比较下去，确保比较时间是一样的，可预防计时攻击。

MessageVerifier 在校验信息的时候就用到了它。
HttpAuthentication Basic 认证的时候就用到了它。
RequestForgeryProtection 在比较 token 的时候也用到了它。

参考

计时攻击原理以及 Rails 对应的防范

railties

结构

1) Railtie

对 Rails 本身的改造。

2) Engine

对 Rails 外围的扩展。

3) Application

初始化时：Bootstrap 在前，Finisher 在后。

与我们应用接头。

实例方法，给 Rails.application 使用。

内容

1) 配置

指的是 Railtie, Engine, Application 的 Configuration.

2) 初始化

"初始化"这里是名词，主要是对它的使用，如 Application 的 Bootstrap 和 Finisher，以及我们项目 AppName 所涉及到的初始化。

3) 启动！

没有额外的"启动"程序，把配置、初始化做好了以后，启动就是自然而然的事了。

4）功能扩展

通过 Railtie、Engine 实现，特别是 Engine 可以大大方便我们组织代码。

继承关系

```
Rails.application.class.ancestors
  => [AppName::Application,
  Rails::Application,
  Rails::Engine,
  Rails::Railtie,
  Rails::Initializable,
  ... ...
  Object,
  ... ...
  Kernel,
  BasicObject]
```

Railtie

Railtie 是 Rails 的核心部分之一。通过它，可以扩展和修改 Rails 的初始化程序。

什么时候需要使用 Railtie? 当你的扩展符合下列情况时，可以考虑：

- 替换默认组件
- Rails 启动时即要配置内容
- Rails 启动时即要初始化内容

每一个 Rails 组件(如：ActionMailer, ActionController, ActionView 和 ActiveRecord 等)都属于 Railtie. 因为它们都需要自己的初始化程序。

Railtie 只是配置及初始化文件

Railtie 不属于真正意义上的"代码"，代码已经完成。想把它运用到 Rails 项目里，并且扩展或修改 Rails 的配置、初始化过程，才需要引进 Railtie.

一个 gem 是 Raitie，通常是指它有 raitie.rb (再准确点，有类继承于 Rails::Railtie) … 并不影响它的其它代码。

通常你的项目代码是单独存放的，raitie.rb 只是针对 Rails 项目初始化或配置工作，不推荐把项目代码放到这里。

Engine 和 Application

Engine 是 Railtie 的子类，所以 Railtie 里的方法，由于继承关系，它也可以使用。 Application 是 Engine 的子类，所以 Railtie 里的方法，由于继承关系，它也可以使用。

其它

查看本项目下，所有的 Railtie：

```
Rails.application.send(:ordered_railties)
```

Rails 启动是一个复杂的过程，你不必知道具体在哪一步执行 Railtie 代码。

Railtie 文件下的内容

提供以下实例方法：

initializer

实际上 `initializer` 定义于 Rails::initializable. 它还可以接受 :before 或 :after 做为参数。

Rails::Railtie include 了它，所以对外提供有 `initializer` 方法:

```
class MyRailtie < Rails::Railtie
  # initializer 来源于 Rails::initializable
  initializer "my_railtie.configure_rails_initialization" do
    # 一些初始化代码
  end
end
```

`initializer` 还可将应用做为参数，所以可以这么用:

```
class MyRailtie < Rails::Railtie
  initializer "my_railtie.configure_rails_initialization" do |ap
p|
    app.middleware.use MyRailtie::Middleware
  end
end
```

config

你可以使用 `config` 对象，它在所有 Railtie 和你的应用里是共用的。

```
class MyRailtie < Rails::Railtie
  # 配置使用什么 ORM
  config.app_generators.orm :my_railtie_orm

  # to_prepare 里的内容在生产环境上只执行一次，在开发环境下每个请求都会请
求一遍。
  config.to_prepare do
    MyRailtie.setup!
  end
end
```

> Note: `config` 定义于 Configurable. delegate :config, to: :instance

rake_tasks & generators

继承于 Rails::Railtie 所以有 `rake_tasks` 方法:

```
class MyRailtie < Rails::Railtie
  rake_tasks do
    load "path/to/my_railtie.tasks"
  end
end
```

继承于 Rails::Railtie 所以有 `generators` 方法:

```
class MyRailtie < Rails::Railtie
  generators do
    require "path/to/my_railtie_generator"
  end
end
```

其它

类方法：

```
# 动词，接 block
rake_tasks
console
runner
generators
```

```
configure # 动词

railtie_name

instance
subclasses

abstract_railtie? # 默认是 Rails::Railtie、Rails::Engine 和 Rails::Application
```

实例方法：

```
config # 名词
configure
railtie_namespace
```

部分方法的解释，可以参考已有解释的方法，或参考 Engine 里的方法。

Initializable

指的是 Rails::Initializable.

本模块被 Railtie 所引用，又由于继承关系，Engine、Application、AppName 都可用里面的方法。
如：自定义的 Railtie 及其子类里常用到的 `initializer` 方法，就是它定义的。

类方法：

```
initializer

initializers_for

initializers
initializers_chain
```

其中，`initializer` 负责创建 initializer，并加入 initializers.

可用 `initializers_for` 获取应用里某类 initializer 的名字：

```
Rails.app_class.initializers_for("web_console").map &:name
```

实例方法：

```
initializers

run_initializers
```

可用 `initializers` 获取应用里所有的 initializer 的名字：

```
Rails.application.initializers.map &:name
```

Initializer

所谓的"初始化"，即便它是一个行为，也要结构化，在 Rails 里用 Initializer 表示。

每一个"初始化"操作，都对应其一个实例对象。

Configuration

Railtie、Engine、Application 都有自己的 Configuration 模块。

由于 Ruby 的继承机制，我们常用的 `config` 可以看作是它们中任意一个的实例对象。所以，理论上来说，它们提供的接口 `config` 都可直接调用。

对外提供接口

```
app_generators
app_middleware

eager_load_namespaces

to_prepare
to_prepare_blocks

watchable_dirs
watchable_files

before_eager_load
before_configuration

after_initialize
before_initialize
```

另，自定义的 Railtie 和自定义的 Engine，也可以对外提供 `config` 接口。

定制自己的 Railtie

一，继承于 Rails::Railtie

继承于 Rails::Railtie，即可创建自己的 Railtie. 在 Rails 启动的时候，它也会被执行。

举例：

```
# lib/my_gem/railtie.rb
module MyGem
  class Railtie < Rails::Railtie
  end
end

# lib/my_gem.rb
require 'my_gem/railtie' if defined?(Rails)
```

创建 Railtie，除了 lib/my_gem/railtie.rb 里继承于 Rails::Railtie 外，你不需要更改其它地方的代码。并不影响 my_gem 原来要做的事，也正如此，my_gem 也可以在 Rails 之外使用。

二，编写自己的 my_railtie/railtie.rb 文件内容

可用 Rails::Railtie 提供的方法。

三，编写 Railtie 内容

做了上述两步后，就是编写内容。该做什么事，做什么事；该完成什么功能，完成什么功能。

四，在应用启动时自动调用定制的 Railtie

自动运行 my_railtie/railtie.rb 里面的相关启动代码。

Railtie 补充

TODO

Rails 默认组件都是 Railtie

Action Mailer

初始化：

```
logger

set_configs

compile_config_methods

show_previews
```

Abstract Controller

引入 **Route** 相关的 **helper**(这里只是调用，定义在 RouteSet 里)。

`routes.rb` 里定义的每一个路由规则都会有对应的 x_url 和 x_path 等 helper 方法可用，这里 include 了这些 helper.

然后，Action Controller 和 Action Mailer 的 Railtie 又 extend Routes Helpers，所以可用。

> Note: 可以通过 `include Rails.application.routes.url_helpers` 然后调用和 Routing 相关的 helper 方法。

Action Controller

initializer

```
Assets config     # 配置 assets_dir，默认是 public/
Set helpers path # 默认是 app/helpers/
Parameters config
Set configs
Compile config methods
```

配置可通过以下方式查看：

```
Rails.configuration.action_controller

或

Rails.application.config.action_controller
```

默认配置项：

```
Rails.configuration.action_controller.keys
# => [:perform_caching, :assets_dir, :logger, :cache_store, :jav
ascripts_dir,
#     :stylesheets_dir, :asset_host, :relative_url_root]
```

Action Dispatch

初始化 configure 及其它。

Action View

```
embed authenticity token in remote forms.

logger.

set configs.

caching.

setup action pack.
```

Active Model

加载 Action Model 相关 I18n

```
ActiveSupport.on_load(:i18n) do
  I18n.load_path << File.dirname(__FILE__) + '/active_model/loca
le/en.yml'
end
```

Active Record

获取 database.yml 的配置信息：

```
Rails.application.config.database_configuration
```

... ...

Active Job

```
logger
set_configs
set_reloader_hook
```

另，配置默认 queue_adapter 由默认的 :inline 改为了 :async

Active Support

```
active_support.deprecation_behavior
active_support.initialize_time_zone
active_support.initialize_beginning_of_week
active_support.set_configs
```

I18n

after_initialize 和 before_eager_load 都执行 initialize_i18n

Engine

Engine 下有 Railtie，上有 Application.

```
Your Application
     |
     V
 Application
     |
     V
   Engine
     |
     V
  Railtie
```

在 Engine 里，可以直接使用 Railtie 提供的方法；
在 Application 里，可以直接使用 Engine 提供的方法。

Engine = Ruby Gem + Rails MVC stack elements.

使用 Engine，可以把一个小型的 Rails 项目当成组件，插入到另一个 Rails 项目里。

它可以有自己的 MVC、路由、Helper、Assets、Rake、Generator 与配置、初始化，甚至是 lib、Migration 和测试。

和一般 **gem** 对比，**Engine** 至少有以下特点：

- 按照约定，遵循 Rails 应用的文件、目录结构
- 从 Railtie 继承来的那一套方法，可用于配置、初始化
- 与 Rails 无缝集成

Engine 文件下的内容

对外提供接口

实例方法：

```
app

config

eager_load!

endpoint

env_config

helpers
helpers_paths

load_console
load_generators
load_runner
load_seed
load_tasks

railties

routes
```

类方法：

```
endpoint

find
find_root

isolate_namespace
```

```
# 并且

isolated? & isolated

engine_name & railtie_name
```

其它方法：

```
delegate :middleware, :root, :paths, to: :config
delegate :engine_name, :isolated?, to: :class
```

有哪些 initializer ？

设置 load path
设置 autoload paths
添加 routing paths
添加 locales
添加 view paths
加载 environment config
Append assets path
Prepend helpers path
加载 config initializers
~~Engines blank point~~

Configuration

Railtie、Engine、Application 都有自己的 Configuration 模块。

由于 Ruby 的继承机制，我们常用的 `config` 可以看作是它们中任意一个的实例对象。所以，理论上来说，它们提供的接口 `config` 都可直接调用。

对外提供接口

实例方法：

```
paths
eager_load_paths
autoload_paths
autoload_once_paths

middleware
generators

root
root=
```

`generators` 使用举例：

```
config.generators do |g|
  g.orm             :data_mapper, migration: true
  g.template_engine :haml
  g.test_framework  :rspec
end

# 或

config.generators.colorize_logging = false
```

`paths` 通过它可以查看 Engine 默认已经在用的路径，当我们定制 Engine 时，按照约定放置文件、目录是个好习惯。

`root`

```
Rails.configuration.root == Rails.root
# => true
```

除 generators、root= 外，其余方法都是 get 获取数据，而不是 set 设置数据。

另，自定义的 Railtie 和自定义的 Engine，也可以对外提供 `config` 接口。

Engine full vs mountable

生成命令 `rails plugin new`

常用参数 `--full` 或 `--mountable`

full

main_app 路由继承于 Engine，所以它们用的路由是同一套。

```
# my_engine/config/routes.rb
Rails.application.routes.draw do
  # ...
end
```

因此，在 main_app 的 config/routes.rb 里不用做任何配置，它们是互通的。

main_app 会继承 Engine 的 model、controller、routes 等。

也就是说它们的环境是一样的。

mountable

首先，从路由上讲它们是隔离开的：

```
# my_engine/config/routes.rb
MyEngine::Engine.routes.draw do
  # ...
end

# parent_app/config/routes.rb
ParentApp::Application.routes.draw do
  mount MyEngine::Engine => "/engine", :as => "namespaced"
end
```

其次，Engine 使用自己的命名空间：

```
# my_engine/lib/my_engine/engine.rb
module MyEngine
  class Engine < Rails::Engine
    isolate_namespace MyEngine
  end
end
```

再者，Engine 创建的表有 engine_name 做为前缀。

model、controller、routes 等都是相互隔离的。

也就是说它们的环境不是一样的。

推荐使用 mountable

`--full` 仅做文件、目录上的分隔，实际上我们没必要使用，有其它方式实现(如：使用命名空间)。

`--mountable` 才是推荐做法(从 Engine 存在的意义及文档上，可以看出这点)。

定制自己的 Engine

Engine = Ruby gem + Rails MVC stack elements.

一，创建自己的 Engine

可用命令 `rails plugin new` 创建自己的 Engine.
常用可选参数 `--full` 或 `--mountable`

两者之间的区别，可以参考【Engine full vs mountable】章节。

拆分一下，步骤大概如下：

1) 继承于 Rails::Engine，一般把它们放在 lib/ 目录下。

```
# lib/my_engine.rb
module MyEngine
  class Engine < Rails::Engine
    # ... ...
  end
end
```

2) 在 config/application.rb (或 Gemfile) 里加载本文件。

Engine 相关的 model、controller 和 helper 会被加载到 app/ 里，route 会被加载到 config/routes.rb, locale 会被加载到 config/locales, tasks 会被加载到 lib/tasks.

```
# config/application.rb
require 'my_engine/engine'

# 或者

# Gemfile
gem 'my_engine', path: "/path/to/my_engine"
```

3) 在 routes.rb 里 mount MyEngine::Engine

```
Rails.application.routes.draw do
  mount MyEngine::Engine => "/engine"
end
```

二，编写 my_engine/engine.rb 文件内容

每个 Engine 都会有自己的 engine.rb 文件。里面有自己的 Engine 类，它继承于 ::Rails::Engine

1) 常用方法之 config、initializer

在这文件里，你可以使用 config, initializer 等方法。这点和定制 Railtie 类似，但不同点是：当前 Engine 的配置和初始化，作用域仅限于当前 Engine.

```
class MyEngine < Rails::Engine
  # 添加新的、额外的目录到 autoload_paths 里
  config.autoload_paths << File.expand_path("../lib/some/path", __FILE__)

  initializer "my_engine.add_middleware" do |app|
    app.middleware.use MyEngine::Middleware
  end
end
```

config 是个方法，但同时它也是 Configuration 的实例对象，所以你可以使用 `config.generators`：

```
class MyEngine < Rails::Engine
  config.generators do |g|
    g.orm             :active_record
    g.template_engine :erb
    g.test_framework  :test_unit
  end
end
```

还可使用 `config.app_generators`：

```
class MyEngine < Rails::Engine
  # note that you can also pass block to app_generators in the s
ame way you
  # can pass it to generators method
  config.app_generators.orm :datamapper
end
```

2) 常用方法之 isolate_namespace

默认 Engine 和应用是在一个环境里的，这意味着应用所有 helper 和命名路由都可以在 Engine 里使用。

你可以使用 `isolate_namespace` 更改此项默认配置，将 Engine 和应用隔离出来。使用举例：

```
module MyEngine
  class Engine < Rails::Engine
    isolate_namespace MyEngine
  end
end
```

此时 `MyEngine` 和应用是隔离了的，假设 MyEngine 有代码：

```
module MyEngine
  class FooController < ActionController::Base
  end
end
```

此时 FooController 仅能使用 `Engine` 里提供的 helper，以及 `MyEngine::Engine.routes` 提供的 url helper.

另外一个改变就是 Engine 里的路由不必再使用 namespace，举例：

```
MyEngine::Engine.routes.draw do
  resources :articles
end
```

resources :articles 自动对应着 `MyEngine::ArticlesController` . 并且不必使用长长的 url helper，例如 `my_engine_articles_path` 可以直接使用 `articles_path`

不受 isolate_namespace 影响的就是对于 model 的调用，仍然使用 engine_name 做为前缀。例如以下例子的 `MyEngine::Article`

```
polymorphic_url(MyEngine::Article.new) # => "articles_path"

form_for(MyEngine::Article.new) do
  text_field :title
  # => <input type="text" name="article[title]" id="article_titl
e" />
end
```

另一个改变是对表名的更改。默认使用 engine_name (在这里是 "my_engine")做为表前缀，也就是说 MyEngine::Article 对应的表名应该是 my_engine_articles

3) 常用方法之 paths

Engine 默认都有自己的文件、目录结构，如果你没有定制，那么就使用默认的：

```
"app",                  eager_load: true, glob: "*"
"app/assets",           glob: "*"
"app/controllers",      eager_load: true
"app/helpers",          eager_load: true
"app/models",           eager_load: true
"app/mailers",          eager_load: true
"app/views"

"app/controllers/concerns", eager_load: true
"app/models/concerns",      eager_load: true

"lib",                  load_path: true
"lib/assets",           glob: "*"
"lib/tasks",            glob: "**/*.rake"

"config"
"config/environments", glob: "#{Rails.env}.rb"
"config/initializers", glob: "**/*.rb"
"config/locales",      glob: "*.{rb,yml}"
"config/routes.rb"

"db"
"db/migrate"
"db/seeds.rb"

"vendor",               load_path: true
"vendor/assets",        glob: "*"
```

`paths` 通过它，可以更改默认的文件、目录结构。

举例，你想把 controller 文件放到 lib/ 目录下：

```
class MyEngine < Rails::Engine
  paths["app/controllers"] = "lib/controllers"
end
```

再或者，controller 在 app/ 和 lib/ 下都可接受：

```
class MyEngine < Rails::Engine
  paths["app/controllers"] << "lib/controllers"
end
```

Application 在 Engine 之上，它又有自己的配置和初始化。它配置了 app/ 下的文件、目录会被自动加载，所以像 app/services 会被自动加载。

4) 常用方法之 endpoint

Engine 内容也可以是一个 Rack Application. 当你的代码本身是 Rack Application，而又想使用 Engine 的特性时，可以这么做：

1) 在自己定义的 Engine 里，使用 `endpoint` ：

```
module MyEngine
  class Engine < Rails::Engine
    # Engine 的内容就是 MyRackApplication
    endpoint MyRackApplication
  end
end
```

2) 和平常一样，在 route 里 `mount` 你的 Engine:

```
Rails.application.routes.draw do
  mount MyEngine::Engine => "/engine"
end
```

5) 常用方法之 middleware

Engine 内容也可以是一个 Middleware. 当你的代码本身是 Middleware，而又想使用 Engine 的特性时，可以这么做：

```
module MyEngine
  class Engine < Rails::Engine
    # Engine 的内容就是 SomeMiddleware
    middleware.use SomeMiddleware
  end
end
```

6) 常用方法之 engine_name

用几个场景可能会用到 engine name:

- routes: 当你使用 mount(MyEngine::Engine => '/my_engine')
- rake task: 当你使用 my_engine:install:migrations

Engine name 默认根据类名而来，如 `MyEngine::Engine` 对应有 `my_engine_engine` . 你可以使用 `engine_name` 进行自定义:

```
module MyEngine
  class Engine < Rails::Engine
    engine_name "my_engine"
  end
end
```

三，编写 Engine 内容

做了上述两步后，就是编写内容。该做什么事，做什么事；该完成什么功能，完成什么功能。

四，在 main_app 引入定制的 Engine

也就是：

在 config/application.rb (或 Gemfile) 里加载本文件。
在 routes.rb 里 mount MyEngine::Engine

其它

mount as - 在 Engine 之外使用其路由

mount(挂载)后 Engine 和应用之间的路由仍然是独立的，你仍然不能在应用里直接使用 Engine 里面的路由。举例：

```
# config/routes.rb
Rails.application.routes.draw do
  mount MyEngine::Engine => "/my_engine", as: "my_engine"
  get "/foo" => "foo#index"
end
```

Engine 外面，使用 `my_engine` 访问 **Engine** 里面的路由：

```
class FooController < ApplicationController
  def index
    my_engine.root_url # => /my_engine/
  end
end
```

Engine 里面，使用 `main_app` 访问 **Engine** 外面的路由：

```
module MyEngine
  class BarController
    def index
      main_app.foo_path # => /foo
    end
  end
end
```

`:as` 可选项默认使用的是 `engine_name` ，所以通常你可以省略它。

还有一种情况，需要传递 engine_name 以便生成路由，举例：

```
form_for([my_engine, @user])
```

这里生成的路由规则类似 `my_engine.user_path(@user)`

helper - Isolated engine's helpers

有时候，你的 Engine 是 Isolated，但你又想使用 Engine 里面定义的 helper，你可以引入某个模块：

```
class ApplicationController < ActionController::Base
  helper MyEngine::SharedEngineHelper
end
```

或者，引入 Engine 里面所有的 helper 模块：

```
class ApplicationController < ActionController::Base
  helper MyEngine::Engine.helpers
end
```

> Note: 这里引入的只是 helpers 目录下的文件，在 Controller 里定义，然后使用 helper_method 的方法不包含在内。

Migrations & seed data

Engine 也可以有自己的迁移文件，和普通应用一样，它们位于 `db/migrate` 下面。

你可以运行以下命令，将 Engine 里的迁移文件复制到应用里：

```
rake ENGINE_NAME:install:migrations
```

Engine 也可以有自己的 seed 文件，它们位于 `db/seeds.rb` 下面。

你可以运行以下命令，将 Engine 里的 seed 文件复制到应用里：

```
MyEngine::Engine.load_seed
```

改变加载顺序，优先加载

场景

```
- app
  - views
    - shared
      - _header.html.erb       <-- 实际渲染的却是这个
  - ...
- config
- ...
- vendors
  - plugins
    - myplugin
      - app
        - views
          - controller1
            - action1.html.erb <-- 在这里渲染
          - shared
            - _header.html.erb <-- 希望渲染的是这个

<%= render 'shared/header' %>
```

按照直观的理解，渲染的应该是第 2 个模板。但实际情况却不是，所以需要我们配置。

你可以使用 `config.railties_order` 改变 Engine 以及应用的加载顺序：

```
# load Blog::Engine with highest priority, followed by applicati
on and other railties
config.railties_order = [Blog::Engine, :main_app, :all]
```

main_app 表示我们的项目本身，在 Application::Finisher 里定义，all 表示所有其它的 Railtie，在 Application::Configuration 里初始化时定义。

> Note: 上面的例子，你也可以用其它手段完成，如 namespace 等。

迁移文件

```
rake my_engine:install:migrations
```

提示

除了 initializer 外，rake_tasks 也会被复制到 main_app 里。所以，有时候你会看到提示：
"Copy migrations from #{railtie_name} to application"

Application

Application 继承于 Engine，负责协调整个启动过程，包括：配置、初始化。

配置

除了和 Engine、Railtie 有一样的配置项外，它新增了自己的配置项，如：cache_classes、consider_all_requests_local、filter_parameters、logger 等。

和我们的配置直接相关：

```
Rails.configuration == Rails.application.config
 => true

Rails.configuration.class
 => Rails::Application::Configuration
```

初始化

Application 负责执行所有 Railtie 和 Engine 的初始化任务。可分为前期准备任务 Bootstrap，和后期收尾任务 Finisher.

Application 文件下的内容

实例方法：

```
# 封装上一级的同名方法
console
generators
rake_tasks
runner
initializer

isolate_namespace

key_generator
message_verifier

reload_routes!

initialized?

config_for

helpers_paths

find_root
```

```
env_config

secrets
```

类方法：

```
create
```

除了以上对外提供的接口外，它还有一些很有用的方法。但不属于对外提供接口，如：

```
initialize!
```

明确使用到的其它类和模块：

```
require 'active_support/key_generator'
require 'active_support/message_verifier'
require 'rails/engine'

autoload :Bootstrap
autoload :Configuration
autoload :DefaultMiddlewareStack
autoload :Finisher
autoload :Railties
autoload :RoutesReloader
```

Bootstrap 打前锋

include by Application.

~~加载 environment hook~~
加载 active support
设置 eager load
初始化 logger
初始化 cache
初始化 dependency mechanism
~~Bootstrap hook~~

Finisher 收尾

include by Application.

添加 generator templates
Ensure autoload once paths as subset
添加 builtin route
构建 middleware stack
定义 **main_app helper**(使用 Engine 时，一个比较重要的概念)
添加 to prepare blocks
运行 prepare callbacks
Eager load!
~~Finisher hook~~
设置 routes reloader hook
设置 clear dependencies hook

另，
`main_app` 可以让你确保使用的是主应用所在的环境，当你使用了 mountable Engine 时，这点可能很重要。

Configuration

Railtie、Engine、Application 都有自己的 Configuration 模块。

由于 Ruby 的继承机制，我们常用的 `config` 可以看作是它们中任意一个的实例对象。所以，理论上来说，它们提供的接口 `config` 都可直接调用。

对外提供接口：

```
attr_accessor :allow_concurrency, :asset_host, :assets, :autoflush_log,
              :cache_classes, :cache_store, :consider_all_requests_local, :console,
              :eager_load, :exceptions_app, :file_watcher, :filter_parameters,
              :force_ssl, :helpers_paths, :logger, :log_formatter, :log_tags,
              :railties_order, :relative_url_root, :secret_key_base, :secret_token,
              :serve_static_assets, :ssl_options, :static_cache_control,
              :session_options, :time_zone, :reload_classes_only_on_change,
              :beginning_of_week, :filter_redirect, :x

attr_writer :log_level
attr_reader :encoding
```

有以下方法：

```
annotations

colorize_logging

database_configuration

log_level

paths

session_store
```

`paths` 除了 Engine 包含的文件、目录结构外，这里有：

```
paths.add "config/database",    with: "config/database.yml"
paths.add "config/secrets",     with: "config/secrets.yml"
paths.add "config/environment", with: "config/environment.rb"
paths.add "lib/templates"
paths.add "log",                with: "log/#{Rails.env}.log"
paths.add "public"
paths.add "public/javascripts"
paths.add "public/stylesheets"
paths.add "tmp"
```

另，自定义的 Railtie 和自定义的 Engine，也可以对外提供 `config` 接口。

Default Middleware Stack

配置 Rails 项目默认的 middleware stack.

相关方法为 `build_stack` ，在 Finisher 里有调用到。

可以通过以下命令查看：

```
Rails.application.send(:default_middleware_stack)
```

如下：

```
Rack::Sendfile

ActionDispatch::Static
ActionDispatch::LoadInterlock

Rack::Runtime
Rack::MethodOverride

ActionDispatch::RequestId

Rails::Rack::Logger

ActionDispatch::ShowExceptions
ActionDispatch::DebugExceptions
ActionDispatch::RemoteIp
ActionDispatch::Reloader
ActionDispatch::Callbacks
ActionDispatch::Cookies
ActionDispatch::Session::CookieStore
ActionDispatch::Flash

Rack::Head
Rack::ConditionalGet
Rack::ETag
```

Rails 应用启动过程

1) 入口 config.ru

```
require ::File.expand_path('../config/environment',  __FILE__)
```

2) 转入 environment.rb

```
require File.expand_path('../application', __FILE__)
```

3) 转入 application.rb

```
require File.expand_path('../boot', __FILE__)
```

4) 转入 boot.rb

```
ENV['BUNDLE_GEMFILE'] ||= File.expand_path('../../Gemfile', __FILE__)
```

5) Gemfile

```
gem 'gem_name'
```

6) 回到 boot.rb

执行 bundle

7) application.rb

7.1) require 'something' # like require 'rails'

7.2) AppName::Application < Rails::Application

7.3) config your AppName! 在上一步继承之后就已经有配置了。

8) enviroment.rb

AppName::Application.initialize!

```
# Initialize the application passing the given group. By default
, the
# group is :default
def initialize!(group=:default) #:nodoc:
  raise "Application has been already initialized." if @initiali
zed
  run_initializers(group, self)
  @initialized = true
  self
end
```

8.1) Bootstrap

8.2) 默认的 Railtie, Engine, Application

8.3) 定制的 Railtie, Engine

构建 middleware stack.
(Rails.application.send :middleware 查看 middleware, 顺序从前到后)
(Rails.application.send :default_middleware_stack 查看 middleware, 顺序是默认)

8.4) Finisher

启动过程基本上都在：

```
Rails::Application - config/application.rb

Rails::Railtie::Configuration - configuration.rb
```

config/application.rb 里先 require 再 config 最后 eager_load.

相关代码：

```
AppName.initialize!
```

```
run_initializers(group, self)


initializers.tsort_each do |initializer|
  initializer.run(*args) if initializer.belongs_to?(group)
end


@initializers ||= self.class.initializers_for(self)

Collection.new(initializers_chain.map { |i| i.bind(binding) })

def initializers_chain
  initializers = Collection.new
  ancestors.reverse_each do |klass|
    next unless klass.respond_to?(:initializers)
    initializers = initializers + klass.initializers
  end
  initializers
end


def run(*args)
  @context.instance_exec(*args, &block)
end
```

各样的钩子，如：

```
before_configuration

before_eager_load

before_initialize

after_initialize
```

它们在 Railtie::Configuration 里定义，使用范围很广。

最后才是 initializer!

Application 补充

TODO

Routes Reloader

充分运用了 File Update Checker，当"路由"相关文件有改动时，可以实现自动重新加载。

开发环境下，不用重启，重新加载路由：

```
Rails.application.reload_routes!
```

相关代码：

```
def reload!
  clear!
  load_paths
  finalize!
ensure
  revert
end
```

这部分，更多信息可以查看"Active Support autoload 的类和模块"下面的【File Update Checker】章节。

由 rake 和 rails 两部分组成。Rails 5 以后这两个命令统一成了 rails，但 rake 仍然可以使用。

rake

tasks.rb 及 tasks 目录

都有单独的 rake 文件， `rake -T` 包含但不限于这下面的命令：

annotations(notes)
initializers
framework(rails)
log
middleware
misc(secret、about、time)
routes
tmp
rails(update、template)
restart
engine(app、db)
statistics(stats)

- Code Statistics
- Code Statistics Calculator

Rakefile 里的 `Rails.application.load_tasks` 会加载本应用使用到第三方 Railtie 和 Engine，及 Rails 自身(按照约定 lib/tasks 也包含在内)所定义的 rake 任务。

可以只查看和某一命名空间有关的 rake 任务，例如和 notes 相关的任务：

```
$ rake -T notes

rake notes
rake notes:custom
```

Source Annotation Extractor

和上面列举的：
rake notes
rake notes:optimize 等相关。

api

rake rdoc (Rails 自身的 API)

test_unit

rake test

还可再细分为 models、helpers、controllers、mailers、integration 和 jobs 等。

rails

commands

`rails` 可接以下命令：

application
console
dbconsole
destroy
generate
plugin
runner
server
update

commands_tasks

常用的有:

- generate
- console
- server
- dbconsole
- new

以及:

- destroy
- plugin new
- runner

> Note: 迁移相关在 Active Record 里的 databases.rake 里定义。

rails console 里的小技巧

1) `app`
会话实例，可用于集成测试。

通过它可以做一些你在集成测试里要做的事。

```
app.class
=> ActionDispatch::Integration::Session

# 创建过程, 类似这样
new_app = Rails.application
session = ActionDispatch::Integration::Session.new(new_app)

session.class
=> ActionDispatch::Integration::Session
```

- 所有的 path 和 url (这里不包含其它 helper)
- request.methods
- response.methods
- 相关的 asset_.methods
- ActionController::Base 引入的 private 等方法

2) `new_session`
返回一个新的集成测试会话实例

3) `reload!`
重新加载环境。这个应该很熟悉 这个最常用，当修改了 env 配置的时候，需要重新加载。使用这个方法可以不必退出后在启动。

4) `helper`
ActionView::Base 实例。 通过它可以直接使用 view 的方法等，例如：所有 helper 方法(这里不包含 path 和 url)。

```
module UsersHelper
  def user_help_1(user_name)
    puts "I am June!"
  end

  def user_help_2
    puts "I am June-Lee!"
  end
end

# 原理类似
helper.send :extend, UsersHelper
```

5) `controller`
ApplicationController 实例。 可以直接使用 ApplicationController 的方法等。

其它

以下有的内容并非 Rails 专有，但很实用，一并列出。

- method 的使用

```
# 方法定义的地方
x.method(:method_name).source_location

# 方法内容
x.method(:method_name).source

# 方法前面的注释说明
x.method(:method_name).comment

# x 可以是对象、类、模块等
```

- 在执行语句后面加分号(;)，可去掉 console 默认的打印输出。
- 下划线(_)可以显示上条正确命令执行的结果。
- Console 清屏快捷键"Ctrl + L"或"Command + K"

- 定制自己的 **console** 方法

举例，我们使用 gem 'factory_girl'，在控制台里为了少敲几个字母，我们用 `fg` 代替 FactoryGirl，可以这么做：

```
# config/environments/test.rb
module Rails
  module ConsoleMethods
    def fg
      @fg ||= FactoryGirl
    end
  end
end
```

- 使用 Pry 代替 IRB

```
# config/application.rb
config.console do
  require 'pry'
  config.console = Pry
end
```

(如果使用 gem 'pry-rails', 默认已经用 pry 代替 irb)

- 使用 y _

格式化重新输出上次内容。

路径 - Root 和 Path

包括 Root & Path，但 Root 已经封装了 Path, 并且只有 Root 对外提供接口。

Root

```
Rails.application.paths
```

提供方法：

```
add

all_paths

autoload_once
autoload_paths
eager_load
load_paths
```

和

```
keys
values
values_at

[]
[]=
```

这部分内容，偏底层了。

Path

Path 元编程提供的几个方法，也挺有用的：

```
%w(autoload_once eager_load autoload load_path).each do |m|
  class_eval <<-RUBY, __FILE__, __LINE__ + 1
    def #{m}!          # def eager_load!
      @#{m} = true     #   @eager_load = true
    end                # end
                       #
    def skip_#{m}!     # def skip_eager_load!
      @#{m} = false    #   @eager_load = false
    end                # end
                       #
    def #{m}?          # def eager_load?
      @#{m}            #   @eager_load
    end                # end
  RUBY
end
```

另，Path 行为和数组有点类似，并且它还 `include Enumerable` ，所以部分操作也可用。

提示信息页面

ApplicationController

这里指的是 Rails 自带的 Rails::ApplicationController, 不是我们 Controller 里继承的 Application.
它和 Application 没有对应关系，只是下面几个 Controller 的父类。

WelcomeController

新建 Rails 项目，没有任何路由及内容时，默认显示的首页 index.

MailersController

邮件预览的列表 index 和详情 preview 页。

InfoController & Info

项目信息页面，包括 Rails、Ruby、Rack 版本等。
有页面：index(同 routes) 和 properties.
默认首页里点击" About your application’s environment" 可查看其内容。

其它

TODO

Rails 文件下的内容

这里指的是 class Rails. 因为它与框架同名，所以这里会加大它的重要指数，下面就说说我们常用的几个类方法。

类方法	解释
application	代表我们的应用，它是 AppName::Application 的实例对象
configuration	我们应用的配置相关信息，它是 Rails::Application::Configuration 的实例对象
env	我们应用所处环境，它是 ActiveSupport::StringInquirer 的实例对象
root	获取项目 root 路径
cache	缓存实例对象
logger	日志实例对象
backtrace_cleaner	它是 Rails::BacktraceCleaner 的实例对象
public_path	获取项目 public/ 路径

Configuration Middleware Stack Proxy

我们常用的 `config.middleware` ，其实是 Middleware Stack Proxy 的实例对象。对应 Rails::Configuration::MiddlewareStackProxy

被 Railtie 所调用，又由于继承关系，Engine、Application、YourApp 都可用。

实例方法：

`use`

```
config.middleware.use Magical::Unicorns
```

`insert_before & insert`

```
config.middleware.insert_before ActionDispatch::Head, Magical::Unicorns
```

`insert_after`

```
config.middleware.insert_after ActionDispatch::Head, Magical::Unicorns
```

`swap`

```
config.middleware.swap ActionDispatch::Flash, Magical::Unicorns
```

`delete`

```
config.middleware.delete ActionDispatch::Flash
```

除以上方法外，还有：

`unshift`

运用上述方法，我们可以增加、删除 Middleware，或改变 Middleware 加载顺序。

AppName, Application, Engine, Railtie

从上层到底层。

```
AppName < Application < Engine < Railtie
```

最直观的就是文件、目录结构，以及配置文件。其次，是默认组件。

默认组件都是 Railtie

active_record, action_controller, action_view, action_mailer, rails/test_unit, sprockets 还有 active_model 都属于 Railtie.

查看配置有哪些 eager_load_namespaces

```
Rails.configuration.eager_load_namespaces

=> [ActiveSupport,
 ActionDispatch,
 ActiveModel,
 ActionView,
 ActionController,
 ActiveRecord,
 ActionMailer,

 Coffee::Rails::Engine,
 Jquery::Rails::Engine,
 Turbolinks::Engine,
 WebConsole::Engine,

 YourAppName::Application]
```

这部分，更多信息可以查看"Railtie"下面的【Configuration】章节。

查看应用有哪些 Initializer

```
Rails.application.initializers.map &:name

=> [:load_environment_hook,
 :load_active_support,
 :set_eager_load,
 :initialize_logger,
 :initialize_cache,
 :initialize_dependency_mechanism,
 :bootstrap_hook,
 # 以上来自 Bootstrap

 "active_support.deprecation_behavior",
 "active_support.initialize_time_zone",
 "active_support.initialize_beginning_of_week",
 "active_support.set_configs",
 # 以上来自 Active Support

 "action_dispatch.configure",
 # 以上来自 Action Dispatch

 "active_model.secure_password",
 # 以上来自 Active Model

 "action_view.embed_authenticity_token_in_remote_forms",
 "action_view.logger",
 "action_view.set_configs",
 "action_view.caching",
 "action_view.setup_action_pack",
 # 以上来自 Action View

 "action_controller.assets_config",
 "action_controller.set_helpers_path",
 "action_controller.parameters_config",
 "action_controller.set_configs",
 "action_controller.compile_config_methods",
 # 以上来自 Action Controller

 "active_record.initialize_timezone",
 "active_record.logger",
```

```
"active_record.migration_error",
"active_record.check_schema_cache_dump",
"active_record.set_configs",
"active_record.initialize_database",
"active_record.log_runtime",
"active_record.set_reloader_hooks",
"active_record.add_watchable_files",
# 以上来自 Active Record

"global_id",
# 以上来自 gem 'globalid'
# Active Job 依赖于它

"active_job.logger",
"active_job.set_configs",
# 以上来自 Active Job

"action_mailer.logger",
"action_mailer.set_configs",
"action_mailer.compile_config_methods",
# 以上来自 Action Mailer

:setup_sass,
:setup_compression,
# 以上来自 gem 'sass-rails'

:jbuilder,
# 以上来自 gem 'jbuilder'

:set_load_path,
:set_autoload_paths,
:add_routing_paths,
:add_locales,
:add_view_paths,
:load_environment_config,
:append_assets_path,
:prepend_helpers_path,
:load_config_initializers,
:engines_blank_point,
# 以上来自 Engine
```

```
# 来自 Engine，略...

:turbolinks,
# 以上来自 gem 'turbolinks'

"web_console.initialize_view_helpers",
"web_console.add_default_route",
"web_console.process_whitelisted_ips",
"web_console.process_command",
"web_console.process_colors",
# 以上来自 gem 'web_console'

# 来自 Engine，略...

:initialize_dependency_mechanism,
# 以上来自 Bootstrap

:add_generator_templates,
:ensure_autoload_once_paths_as_subset,
:add_builtin_route,
:build_middleware_stack,
:define_main_app_helper,
:add_to_prepare_blocks,
:run_prepare_callbacks,
:eager_load!,
:finisher_hook,
:set_routes_reloader_hook,
:set_clear_dependencies_hook]
# 以上来自 Finisher
```

这部分，更多信息可以查看"其它"对应的【Initializable】章节。

Backtrace Cleaner

是运用，而不是定义，定义于 ActiveSupport::BacktraceCleaner.

使用类似：

```
bc = BacktraceCleaner.new

# 定义：(部分)删除内容里的 Rails.root
bc.add_filter   { |line| line.gsub(Rails.root.to_s, '') }

# 定义：(整行)删除 mongrel 或 rubygems 里记录的内容
bc.add_silencer { |line| line =~ /mongrel|rubygems/ }

# 执行
bc.clean(exception.backtrace)
```

Rails 里用 `Rails.backtrace_cleane` 替换上面的 `bc` .

Rack Logger

用于日志记录，默认已经添加到 middleware 堆栈。

Generators

包含了几乎所有 generator 相关知识，参考【Generators】章节。

Generators

把常见的手动操作，用命令来实现。

执行 generator 时，会按先后顺序执行每一个 public 方法。

文件、目录、参数、操作！

使用方式：

```
rails generate GENERATOR [args] [options]
```

如果，你记不住这么多命令，不要紧，按按照手动操作，然后自己实现也是可以的。

该目录下，还有 Rails 自带的 generator 的源代码，它们可供参考。

> Note: 源代码放在 railties 子目录里，所在路径：railties/lib/rails/generators

Thor

Thor 和 rake 类似，提供了功能强大的命令行接口。

因为 Rails 的 generator 实际上是封装了 Thor ，所以还有 Thor Actions

Actions 实例方法：

`action`
封装一个实例对象，并调用它。

`append_to_file`
向文件里追加文本内容。

```
append_to_file 'config/environments/test.rb', 'config.gem "rspec
"'
```

`apply`
加载并执行文件。

```
apply "recipes/jquery.rb"
```

`chmod`
更改文件或目录的权限。

```
chmod "script/server", 0755
```

`comment_lines`
注释指定文件里面符合条件的行。

```
comment_lines 'config/initializers/session_store.rb', /cookie_st
ore/
```

`copy_file`
复制文件。(默认源文件放在 source_root 下)

```
copy_file "README", "doc/README"
```

`create_file & add_file`
创建新文件。

```
create_file "config/apache.conf", "your apache config"
```

`create_link(destination, *args, &block) (also: #add_link)`
创建一个链接文件。

```
create_link "config/apache.conf", "/etc/apache.conf"
```

`destination_root`
返回目标目录。

`destination_root=(root)`
设置目标目录。

`directory`
按规则复制整个目录。(并不是直接复制，注意转换规则)

```
# 源文件、目录如下：
doc/
  components/.empty_directory
  README
  rdoc.rb.tt
  %app_name%.rb

# 复制整个目录：
directory "doc"

# 得到目标文件、目录：
doc/
  components/
  README
  rdoc.rb
  blog.rb
```

`empty_directory`
创建一个空目录。

```
empty_directory "doc"
```

`find_in_source_paths`
在源目录里查找文件。

`get`
获取内容并放到文件里。

```
get "http://gist.github.com/103208", "doc/README"
```

`gsub_file`
按正则替换文件内容。

```
gsub_file 'README', /rake/, :green do |match|
  match << " no more. Use thor!"
end
```

`in_root`
到根目录执行 block 里的代码。

`inject_into_class`
将内容插入到指定的文件，指定的 class 后面。(比 insert_into_file 更精确)

```
inject_into_class "app/controllers/application_controller.rb", A
pplicationController do
  "  filter_parameter :password\n"
end
```

`insert_into_file & inject_into_file`
将内容插入到指定的文件内。(如：添加 js 文件后，一般会在 application.js 里插入 require 等代码)

```
insert_into_file "config/environment.rb",
                 "config.gem :thor",
                 :after => "Rails::Initializer.run do |config|\n"
```

`inside`
在根目录或指定的目录里执行 block 里的代码。

`link_file`
链接文件。

```
link_file "README", "doc/README"
```

`prepend_to_file & prepend_file`
在文件开头处追加文本内容。

```
prepend_to_file 'config/environments/test.rb', 'config.gem "rspe
c"'
```

`relative_to_original_destination_root`
以目标为根目录，获取相对路径。

`remove_file & remove_dir`
移除文件。

```
remove_file 'README'
```

`run`
执行某条命令，并返回执行结果。

```
inside('vendor') do
  run('ln -s ~/edge rails')
end
```

`run_ruby_script`
运行 Ruby 脚本。(照顾 WIN32 的用户)

`source_paths`
得到源路径。(方便后续操作)

`template`
复制示例模板文件，生成新的 ERB 文件。

`thor`
运行 thor 命令。

```
thor :list, :all => true, :substring => 'rails'
```

`uncomment_lines`
去掉指定文件里符合条件的行的注释。

```
uncomment_lines 'config/initializers/session_store.rb', /active_
record/
```

Actions 类方法：

`add_runtime_options!`
添加了 force、pretend、quiet、skip 这几个 class_option.

`source_paths`
存储并返回定义这个类所在的位置

`source_paths_for_search`
获取 source_paths、source_root(有的话)、source_paths(父亲的)

`source_root`
存储并返回定义这个类所在的位置。(Base 已经覆盖此方法)

其它

`argument`
给我们的"命令行"添加参数，并有 attr_accessor
(注意：这里的参数区别于类或方法里的参数、可选参数)
示例：

```
# 这里的 [method method] 这部分
rails g mailer NAME [method method] [options]
```

`ClassName#public_instance_method`
一般的，类名会被当做 namespace，而实例方法会被当做 task，也就是：

```
class_name:public_instace_method
```

(实例方法的参数仍被当做参数对待)

`desc`
对 task 的描述。

`method_option 和 method_options`
使用 task 时传递的可选参数。

`invoke`
调用方法(为什么不直接调用？)

`Thor::Group`
继承于它的话，下面的实例方法会被当成"组"，会按照定义顺序执行。

`class_option 和 class_options`
使用 task 时传递的可选参数。(配合 Thor::Group 很好)

`namespace`
定义 namespace (不使用默认以 ClassName 生成的)

`ClassName.start`
可以把 task 封装成可执行命令，不必用 thor 命令。这是启动命令，通常放在最后。

`option 和 options`
可选参数(如：thor my_cli:hello --from "Carl Lerche" Kelby 这里的 --from)

`public_task & public_command`
定义一个实例方法，方法内容就是其父类的私有方法。

```
public_command :foo
```

链接

[What Is Thor]
[Thor Wiki]

generators 下的目录、文件

对于 Generator, Rails 是自产(定义)和自销(自销)，还对外提供接口，供我们使用。

全部的文件及目录：

```
# 目录
css/
erb/
js/assets/
rails/
test_unit/

# 文件
actions.rb
active_model.rb
app_base.rb
base.rb
erb.rb
generated_attribute.rb
migration.rb
model_helpers.rb
named_base.rb
resouce_helpers.rb
test_case.rb
test_unit.rb

# 目录
actions/
testing/
```

重要的类：

Base、Named Base 以及 App Base.

自己调用自己：

其中，除 actions/ 和 testing/ 外的其它 5 个目录为调用。

```
css/
erb/
js/assets/
rails/
test_unit/
```

我们在写自己的脚本时，可以参考。

方法、别名、选项默认值

```
api_only!

fallbacks

help

hidden_namespaces & hide_namespace & hide_namespaces

invoke

levenshtein_distance

no_color!

print_generators

public_namespaces

sorted_groups

subclasses
```

别名：

```
DEFAULT_ALIASES = {
  rails: {
    actions: '-a',
    orm: '-o',
    javascripts: '-j',
    javascript_engine: '-je',
    resource_controller: '-c',
    scaffold_controller: '-c',
    stylesheets: '-y',
    stylesheet_engine: '-se',
    scaffold_stylesheet: '-ss',
    template_engine: '-e',
    test_framework: '-t'
  },

  test_unit: {
    fixture_replacement: '-r',
  }
}
```

选项默认值：

```
DEFAULT_OPTIONS = {
  rails: {
    api: false,
    assets: true,
    force_plural: false,
    helper: true,
    integration_tool: nil,
    javascripts: true,
    javascript_engine: :js,
    orm: false,
    resource_controller: :controller,
    resource_route: true,
    scaffold_controller: :scaffold_controller,
    stylesheets: true,
    stylesheet_engine: :css,
    scaffold_stylesheet: true,
    test_framework: false,
    template_engine: :erb
  }
}
```

Base

继承于 Thor::Group 并 include Rails::Generators::Actions

类方法

```
base_root
default_source_root
desc
hide!
namespace
remove_hook_for

hook_for

source_root

class_option
```

`source_root` generator 示例模板文件(templates)所在位置。

```
# 默认在 templates 目录下
source_root File.expand_path("../templates", __FILE__)
```

`hook_for` 触发其它的 generator，以 --option 的形式提供。

`desc` 终端里使用对应命令获取帮助(即 --help)会在最后显示此内容。

其它类方法

```
add_shebang_option!
banner
base_name

default_aliases_for_option
default_for_option
default_generator_root
default_value_for_option

generator_name
usage_path
```

实例方法

```
extract_last_module
```

其它

```
filename_with_extensions
```

Actions

主要是生成指定类型的文件，或给已有文件添加内容。

include by Base.

实例方法

```
add_source

after_bundle

environment & application

capify!

gem
gem_group

generate

git

initializer

lib

rake

rakefile

readme

route

vendor
```

其它实例方法

```
extify
log
quote
```

~~Create Migration~~

被 Migration 调用，但应该不能直接使用。

include by Migration.

实例方法

```
existing_migration

exists?
identical?

migration_dir
migration_file_name

relative_existing_migration

revoke!
```

其它实例方法

```
on_conflict_behavior

say_status
```

Named Base

继承于 Base.

默认 `rails g generator` 生成的就是继承于它。

```
check_class_collision

template

application_name

attributes_names

class_name
class_path

file_path

human_name

i18n_scope

indent
index_helper
inside_template
inside_template?

module_namespacing

namespace
namespaced?
namespaced_class_path
namespaced_file_path
namespaced_path

plural_file_name
plural_name
```

```
plural_table_name
pluralize_table_names?

regular_class_path

route_url

singular_name
singular_table_name

table_name

uncountable?

wrap_with_namespace
```

`check_class_collision` 举例 rails generate controller NAME [action action] [options] 检测这里的 NAME 是否已经被使用，也就是说是否有冲突。(因为后续会创建新的文件，如果冲突的话，之前的文件及内容会被覆盖)

```
attr_reader :file_name
```

其它

在外面，只要直接或间接 add_dependency "railties"，即可使用，因为它属于这个(而不是 rails)

- 继承于 NamedBase 或 Base

这是最常见的用法。

- 直接使用某个模块

当然你也可以更进一步，继承于其它模块，但因为没有 API 和文档，使用门槛提高不少。

- 目的很明确
- 非常确定这个模块干的是什么

举例：

Erb::Generators::ControllerGenerator

```
require 'rails/generators/erb/controller/controller_generator'

module Haml
  module Generators
    class ControllerGenerator < Erb::Generators::ControllerGener
ator
      source_root File.expand_path("../templates", __FILE__)

      protected

      def handler
        :haml
      end
    end
  end
end
```

~~Generated Attribute~~

生成过程中的一些属性判断。

~~Erb Generators Base~~

继承于 Rails::Generators::NamedBase

```
formats
format
handler

filename_with_extensions
```

~~App Base~~

继承于 Base.

针对 **Rails** 本身提供了大量方便使用的方法。注意，它们不具备通用性。更多信息，可以查看源代码，这里就不列出了。

```
rails new APP_PATH [options]
```

创建新 **Rails** 应用的时候，它会派上大用场。

```
rails plugin new APP_PATH [options]
```

创建新 **engine**的时候，也会派上大用场。

~~Gemfile Entry~~

```
github
new
path
version
```

上面的方法都不能直接使用。

Generators 文件下的内容

类方法：

```
invoke

fallbacks # 添加 fallback

help # 输出帮助信息

no_color! # 在终端里输出没有颜色

subclasses # generators 的子类
public_namespaces

hidden_namespaces
hide_namespaces & hide_namespace

print_generators

sorted_groups
```

`invoke` 授受 namespace, arguments 和 behavior，它是 generate, destroy 和 update 等命令的入口。

`fallbacks` 使用举例：

```
Rails::Generators.fallbacks[:shoulda] = :test_unit
```

其它类方法：

```
levenshtein_distance
```

其它：

```
DEFAULT_ALIASES = {
  rails: {
    actions: '-a',
    orm: '-o',
    javascripts: '-j',
    javascript_engine: '-je',
    resource_controller: '-c',
    scaffold_controller: '-c',
    stylesheets: '-y',
    stylesheet_engine: '-se',
    template_engine: '-e',
    test_framework: '-t'
  },

  test_unit: {
    fixture_replacement: '-r',
  }
}

DEFAULT_OPTIONS = {
  rails: {
    assets: true,
    force_plural: false,
    helper: true,
    integration_tool: nil,
    javascripts: true,
    javascript_engine: :js,
    orm: false,
    resource_controller: :controller,
    resource_route: true,
    scaffold_controller: :scaffold_controller,
    stylesheets: true,
    stylesheet_engine: :css,
    test_framework: false,
    template_engine: :erb
  }
}
```

```
RAILS_DEV_PATH = File.expand_path("../../../../../..", File.dirn
ame(__FILE__))
RESERVED_NAMES = %w[application destroy plugin runner test]
```

Migration

require 'actions/create_migration'

```
create_migration

migration_template

set_migration_assigns!
```

`migration_template` 和 template、copy_file 类似，复制模板生成新的文件。但不同点在于，这里新生成的文件会自动加上时间戳。

Active Model

功能是：定制 **Controller** 里默认的代码。它只是接口，具体让各个 **ORM** 实现。

included by ResourceHelpers & ModelHelpers

类方法：

```
all
build
find
new
```

实例方法：

```
destroy
errors
save
update
```

Resource Helpers

从 Named Base 里拆分而来。

```
rails generate resource
```

```
rails generate scaffold
```

```
rails generate scaffold_controller
```

方法：

```
attr_reader :controller_name, :controller_file_name
```

和：

```
controller_class_path
controller_file_path
controller_class_name
controller_i18n_scope

orm_class
orm_instance

assign_controller_names!
```

~~Model Helpers~~

为生成起辅助作用。

```
rails generate model
```

~~Generated Attribute~~

```
field_type

default

plural_name
singular_name

human_name

index_name

column_name

foreign_key?

reference?

polymorphic?

required?

has_index?

has_uniq_index?

password_digest?

token?

inject_options

inject_index_options

options_for_migration
```

Rails 里所有的配置项

链接 Configuring Rails Applications

配置项，除了 Configuration 对象外，还可以由 `class_attribute` 进行设置。前者相当于只有一个总开关，只能决定开或关。而后者除了总开关外，还可以有分开关，总开关设置状态后，可以不设置状态（继承），或者设置自己的状态。

附录，查看各个 Railtie 所带配置项：

```
railties = ['action_mailer', 'active_record', 'action_controller'
,
  'action_dispatch', 'action_view', 'active_support', 'i18n', 'a
ssets']

railties.each do |railtie|
  p railtie.camelize
  p "=============="

  klasses = Rails.configuration.send(railtie).values.map(&:class
).uniq

  klasses.each do |klass|
    p klass

    Rails.configuration.send(railtie).each_pair do |k,v|
      p k if v.class == klass
    end
  end
end
```

一般常用配置项

```
# 自动加载目录
config.autoload_paths
# 立即加载目录
config.eager_load_paths

# 立即加载开关。开发环境 false，生产环境 true
# eager_load 始终是立即加载，它只影响 autoload 配置
config.eager_load

# 每次 HTTP 请求之间，是否重新加载环境(类、模块)
# 是，则类似 load; 否，则类似 require
# 影响控制台里的 reload! 效果
config.cache_classes

# 缓存存储方式
config.cache_store

config.console

config.disable_dependency_loading

config.encoding

config.exceptions_app

config.file_watcher

# 网站是否强制启用 https
config.force_ssl

# log 格式处理
config.log_formatter
# debug、还是 info
config.log_level

config.logger
```

```
config.middleware

config.reload_classes_only_on_change

secrets.secret_key_base

# 请使用 `public_file_server.enabled`
config.serve_static_files

# session 存储方式及限制条件
config.session_store

# 时区，默认为 UTC
config.time_zone
```

Action Mailer

FalseClass

```
:raise_delivery_errors
```

String

```
:assets_dir
:javascripts_dir
:stylesheets_dir
:preview_path
```

TrueClass

```
:show_previews
```

NilClass

```
:asset_host
:relative_url_root
```

其它：

```
config.action_mailer.logger

config.action_mailer.smtp_settings

config.action_mailer.sendmail_settings

config.action_mailer.delivery_method

config.action_mailer.perform_deliveries

config.action_mailer.default_options

config.action_mailer.observers

config.action_mailer.interceptors
```

Active Record

TrueClass

```
:maintain_test_schema
:belongs_to_required_by_default
```

其它：

```
config.active_record.logger

config.active_record.primary_key_prefix_type

config.active_record.table_name_prefix
config.active_record.table_name_suffix

config.active_record.schema_migrations_table_name

config.active_record.pluralize_table_names

config.active_record.default_timezone

config.active_record.schema_format

config.active_record.timestamped_migrations

config.active_record.lock_optimistically

config.active_record.cache_timestamp_format

config.active_record.record_timestamps

config.active_record.partial_writes

config.active_record.dump_schema_after_migration
```

Action Controller

FalseClass

```
# 是否启用 Fragment 缓存
:perform_caching
```

String

```
:assets_dir
:javascripts_dir
:stylesheets_dir
```

ActiveSupport::Logger

```
:logger
```

ActiveSupport::Cache::NullStore

```
:cache_store
```

NilClass

```
:asset_host
:relative_url_root
```

TrueClass

```
:forgery_protection_origin_check

# Helper 方法在所有 View 里都可用
:include_all_helpers

per_form_csrf_tokens
```

其它：

```
config.action_controller.asset_host

config.action_controller.default_static_extension

config.action_controller.default_charset

config.action_controller.logger

config.action_controller.request_forgery_protection_token

config.action_controller.allow_forgery_protection

config.action_controller.permit_all_parameters

config.action_controller.action_on_unpermitted_parameters

config.action_controller.always_permitted_parameters
```

Action Dispatch

```
config.filter_parameters
config.filter_redirect

# 决定了报错时，是否直接在 Web 上显示错误堆栈信息
config.consider_all_requests_local
```

NilClass

```
:x_sendfile_header
:default_charset
```

TrueClass

```
:ip_spoofing_check
:show_exceptions
:perform_deep_munge
:always_write_cookie
```

FalseClass

```
:ignore_accept_header
:rack_cache
```

Fixnum

```
:tld_length
```

Hash

```
:rescue_templates
:rescue_responses
:default_headers
```

String

```
:http_auth_salt
:signed_cookie_salt
:encrypted_cookie_salt
:encrypted_signed_cookie_salt
```

Symbol

```
:cookies_serializer
```

Action View

TrueClass

```
:debug_missing_translation
```

其它：

```
config.action_view.field_error_proc provides

config.action_view.default_form_builder

config.action_view.logger

config.action_view.erb_trim_mode

config.action_view.embed_authenticity_token_in_remote_forms

config.action_view.prefix_partial_path_with_controller_namespace

config.action_view.raise_on_missing_translations

# 每一次请求之间，是否需要重新加载模板
# 默认和 cache_classes 设置的一样
config.action_view.cache_template_loading
```

Active Support

Symbol

```
:deprecation
```

其它：

```
config.active_support.bare

config.active_support.test_order

config.active_support.escape_html_entities_in_json

config.active_support.use_standard_json_time_format

config.active_support.time_precision
```

I18n

Array

```
:railties_load_path
:load_path
```

其它：

```
config.i18n.available_locales

config.i18n.default_locale

config.i18n.enforce_available_locales
```

Generators

`config.generators`

```
assets

force_plural

helper

integration_tool

javascripts

javascript_engine

orm

resource_controller

scaffold_controller

stylesheets

stylesheet_engine

test_framework

template_engine
```

Assets

Array

```
:_blocks
:paths
:precompile
:resolve_with
```

String

```
:prefix
:version
```

NilClass

```
:manifest
```

TrueClass

```
:debug
:compile
:digest
:raise_runtime_errors
```

Fixnum

```
:cache_limit
```

其它：

```
# 配置静态资源所在网址。我们网站所用的静态资源可以单独存放
# CDN、网站对静态资源访问（并发）比较大的时候可以考虑
config.asset_host

# 是否使用 Assets Pipeline, 默认为 true
config.assets.enabled

config.assets.compress

config.assets.css_compressor
# 如：config.assets.css_compressor = :yui

config.assets.js_compressor
# 如：config.assets.js_compressor = :uglifier

config.assets.cache_store

config.assets.logger
```

Environments

```
config.cache_classes = true

config.eager_load = true

config.consider_all_requests_local       = false

config.public_file_server.enabled = ENV['RAILS_SERVE_STATIC_FILES'].present?

config.log_level = :debug

config.log_formatter = ::Logger::Formatter.new
```

ache_classes = false 每一次 HTTP 请求之间，都会重新加载代码，类似于 load 方法。 而设置为 true 后，则只会在第一次加载，后续不加载，相当于 require.

```
Rails.configuration.instance_variables

=> [
# 项目所在操作系统上的绝对路径
:@root,

# :rails, :active_record, :test_unit 等
:@generators,

# 用到的中间件
:@middleware,

# 编码（默认是 utf-8）
:@encoding,

# 允许并发？
:@allow_concurrency,

# 本地请求。决定了错误堆栈显示
```

```
:@consider_all_requests_local,

# 日志里过滤什么参数
:@filter_parameters,

# 日志里重定向过滤
:@filter_redirect,

# 辅助文件所在路径
:@helpers_paths,

# 新项目默认首页，是否使用、使用哪个页面
:@public_file_server,

# 强制 https 链接
:@force_ssl,

# https 链接参数
:@ssl_options,

# 怎么存储 session
:@session_store,

# session 相关参数
:@session_options,

# 时区格式。默认用 UTC
:@time_zone,

# 每周开始时间。默认是周一，比如有的是周日
:@beginning_of_week,

# 日志
:@log_level,

# 缓存怎么存、放在哪？
:@cache_store,

# 默认为 all
:@railties_order,
```

```
:@relative_url_root,

# 即使是开发环境，也应该只重新加载更改过的代码
:@reload_classes_only_on_change,

# 文件更改检测
:@file_watcher,

:@exceptions_app,

# 默认 true
:@autoflush_log,

# 默认 Formatter
:@log_formatter,

# 默认 true
:@eager_load,

# 默认没有
:@secret_token,

# 默认没有
:@secret_key_base,

# 是否只用 Rails API
:@api_only,

# 默认 default
:@debug_exception_response_format,

# 自己配置的，可以用这
:@x,

:@paths,

# 自动加载
:@autoload_paths,
```

```
# 延迟加载
:@eager_load_paths,

# 自动加载一次
:@autoload_once_paths,

# 缓存类（代码）
:@cache_classes]
```

设置项补充

```
# 框架初始化后，运行这里的任务
config.after_initialize takes

# 它里面所包含的 namespace 都会被立即加载
config.eager_load_namespaces

# 和 autoload_paths 区别在于：开发环境下，每一次请求之间，它不会被更新了
# 使用它的目录，必须同时使用 autoload_paths
config.autoload_once_paths

# 如果你想把周日当做第一天，那也可以
config.beginning_of_week

# 打印日志里是否显示颜色
config.colorize_logging

# 给 log 打标签
# 子域名或多服务器时可考虑
config.log_tags
```

~~Middleware~~

通过 `config.middleware` 有什么方法可用？
参考【Configuration Middleware Stack Proxy】

有什么 middleware 可供选择？
参考【Action Dispatch Middleware】

其它

一些有意思的点，或者语法糖，或者对于新手来说是"魔术"。

继承心得

C < B < A 有时候之所以 B 要 extend A 并不是为了 B 自己使用，而仅仅是为了方便 C

继承关系及 ancestors

父类和模块都有此方法，子类没有(重写)此方法，执行顺序：模块按被加载的顺序逆序，最后到父类。

```
module A
  def do_something
    puts 'A -> do_something'
    super
  end
end

module B
  def do_something
    puts 'B -> do_something'
    super
  end
end

module C
  def do_something
    puts 'C -> do_something'
    super
  end
end


class Parent
  def do_something
    puts 'Parent -> do_something'
  end
end
```

```
class Child < Parent
  include A
  include B
  include C
end

p Child.ancestors
# => [Child, C, B, A, Parent, Object, Kernel, BasicObject]


Child.new.do_something
# 输出 =>
#
# C -> do_something
# B -> do_something
# A -> do_something
# Parent -> do_something
```

> Note: 以上参考 modules.rb

inherited 方法

`inherited(subclass)`

当前类定义子类时，就会触发此回调。(类似 Module.html#included)

举例:

```
class Foo
  def self.inherited(subclass)
    puts "New subclass: #{subclass}"
  end
end

class Bar < Foo
end

class Baz < Bar
end
```

输出:

```
New subclass: Bar
New subclass: Baz
```

另：继承于一个类，父类的类方法就是子类的类方法，父类的实例方法就是子类的实例方法。

extend 方法

这里专指：一个对象继承一个模块。作用是：给一个对象添加实例方法(会覆盖原有方法)。

```
module Mod
  def hello
    "Hello from Mod.\n"
  end
end

class Klass
  def hello
    "Hello from Klass.\n"
  end
end

k = Klass.new
k.hello         #=> "Hello from Klass.\n"
k.extend(Mod)   #=> #<Klass:0x401b3bc8>
k.hello         #=> "Hello from Mod.\n"
```

ActiveSupport::Cache 里就用这种方式给 Store 实例对象添加了 Local Cache 相关方法。

Rails 源代码里一些常用方法

Active Support

`eager_autoload` 和 `autoload`

`class_attribute`

```
定义一个类属性，子类继承于父类。
子类可以更改自己的属性，但不影响到父类的。
```

`attr_internal`

```
alias_method :attr_internal, :attr_internal_accessor

声明一个读、写属性，功能类似 attr_accessor, 但内部实现有一点点不同。
```

`mattr_accessor`

```
alias :cattr_accessor :mattr_accessor

定义一个类属性，同时具备类和实例级别的读、写。(这里类似全局变量了)
```

`delegate`

```
委托，将它人的方法做为已用。
```

后面是个对象即可，而 Ruby 又号称"一切皆对象"。

最终通过 module_eval 并重新定义了方法。

委托类方法给实例对象使用。

```
class Foo
  def self.hello
    "world"
  end

  delegate :hello, to: :class
end

Foo.new.hello # => "world"
```

`config_accessor`

```
定义一个属性，同时具备类和实例级别的读、写。
实例可以更改自己的属性，但不影响到类的。
```

`define_callbacks`

```
定义回调。
```

`run_callbacks`

```
运行回调。
```

`attr_internal_writer`

```
功能类似 attr_writer，但内部实现有一点不同。
```

`info` `debug` `warn` `error` `fatal` `unknown` 也就是 Rails.logger 的各个级别(或者说类别)。

`included`

```
指的是 ActiveSupport::Concern 所提供的实例方法。
当此模块被 include 时，执行什么代码。
```

`extract_options!`

```
把可选参数里的 Hash 部分，萃取出来。
```

Abstract Controller

`abstract!`

```
声明此 Controller 是抽象的。

ActionController::Metal 和 ActionController::Base 都声明为抽象的。(
作用参考下面解释)
```

我们自定义的 Controller 里的 public instance methods(公开实例方法) 都会被当做 action 来对待。 因此，继承的时候要做一些处理， 以避免父类的实例方法被当做 action. 目前，解决方法是把父类声明为：abstract = true

`helper`

```
引入外部模块(专指 helper 模块)，功能类似 include.
```

`helper_method`

```
将 Controller 里的方法转化为 helper 方法。
```

Railtie

delegate `:config`, to: :instance

Initializable

`initializer`

Rails assets precompile

概述

Assets Pipeline 主要功能 **&** 特性

合并、压缩、解析 css, js 文件。

合并：将多个 js 或 css 文件压缩打包成单一文件，减少 http request 的大小与数量。
压缩：可以去除空格、注释等。
解析：你可直接使用 SCSS 和 CoffeeScript, 它们会被解析成 css 和 js.

主要通过 **Sprockets** 完成

Assets Pipeline 的功能主要由重要的组件 Sprockets 完成。
Sprockets 用来从你的 assets 路径打包压缩你所有的 assets 后包裝成一个文件，然后放到你目的地路径(public/assets).

通过它可以对 css, js 等静态资源进行编译、压缩。

命令行取消，不使用它：

```
rails new appname --skip-sprockets
```

这会使得原来的：

```
require 'rails/all'
```

变成

```
require "rails"
# Pick the frameworks you want:
require "active_model/railtie"
require "active_job/railtie"
require "active_record/railtie"
require "action_controller/railtie"
require "action_mailer/railtie"
require "action_view/railtie"
# require "sprockets/railtie" # <- 重点是这行
require "rails/test_unit/railtie"
```

sprockets 已经被取消掉。

并且，原来的：

```
gem 'sass-rails'
gem 'uglifier'
gem 'coffee-rails'
```

变成：

```
gem 'coffee-rails', '~> 4.1.0'
```

默认的 **3** 个存放静态资源的地方

默认 3 路径：
app/assets 放置我们自己所写的 js、css 或 images, 并且它们和业务关联比较紧密。实际上，这种形式用得最多。
lib/assets 放置我们抽取出来的 assets, 一般来说这样的代码比较通用。实际上，这种形式用得最少。
vendor/assets 是放一些我们从第三方引进的 assets，例如一些 jQuery 插件。

开发环境，你可以把你的 rails app 跑起來后，从 HTML 源代码里查看引入了那些样式或脚本，进而修改，很方便。

这 3 个目录，看似是分开的。但生产环境下，它们后会被编译、打包在一起，也就是说，它们其实是连通的。

.erb 使用 Asset Helper 方法

因为 Sprockets-rails 已经引入了以下两个模块

```
include ActionView::Helpers::AssetUrlHelper
include ActionView::Helpers::AssetTagHelper
```

所以，在 *css.erb 文件里，你都可以使用以下方法：

```
stylesheet_link_tag
javascript_include_tag
```

```
image_tag
```

```
asset_path
asset_data_uri
```

同样的，使用 js.erb 后缀之后可以用 `asset_path` 等方法。

使用举例：

```
audio_path("horse.wav")  # => /audios/horse.wav
audio_tag("sound")       # => <audio src="/audios/sound" />

font_path("font.ttf")    # => /fonts/font.ttf

image_path("edit.png")   # => "/images/edit.png"
image_tag("icon.png")    # => <img src="/images/icon.png" alt="Icon" />

video_path("hd.avi")     # => /videos/hd.avi
video_tag("trailer.ogg") # => <video src="/videos/trailer.ogg" />
```

sass-rails 提供的几个 Asset Helper 方法

由 sass-rails 提供几个以 `-url` 和 `-path` 结尾的 Asset Helpers 方法及结果：

```
image-url("rails.png") # => url(/assets/rails.png)
image-path("rails.png") # => "/assets/rails.png".

asset-url("rails.png") # => url(/assets/rails.png)
asset-path("rails.png") # => "/assets/rails.png"

asset-data-url("rails.png") # => url(data:image/png;base64,iVBOR
w0K...)
```

注意，你可以同时使用 sass-rails 提供的 `-url` 及原生的 `url` 方法；使用 `-url` 时参数带引号，使用 `url` 时参数不带引号。

Sprockets 提供 require 等指令

```
require # 引入某个文件。如果有重复引入，它会自动忽略，只引入一次。
require_directory # 引入某个目录下的文件，不会递归其子目录。
require_tree # 引入某个目录及递归其子目录下的所有文件，默认指的是当前目录。

require_self # 引入在自己文件里写的样式或脚本。

link

depend_on
depend_on_asset

stub
```

上述几个方法，由 Sprockets-rails 提供。

但上述几个方法，不要在 SASS/SCSS 文件里使用，而是使用 sass-rails 提供的 @import 方法。

根据后缀名，决定编译顺序

注意后缀名的顺序：

```
app/assets/javascripts/projects.js.erb.coffee
```

从右到左一个个解析的。

生产环境，需要注意的点

1) Precompiling Assets

```
RAILS_ENV=production bin/rake assets:precompile
```

```
load 'deploy/assets'
```

另，上述会生成、使用 shared/assets 目录。

预编译的文件，默认是：

```
[ Proc.new { |filename, path| path =~ /app\/assets/ && !%w(.js .css).include?(File.extname(filename)) },
/application.(css|js)$/ ]
```

新增：

```
Rails.application.config.assets.precompile += ['admin.js', 'admin.css', 'swfObject.js']
```

另，只使用 .js 或 .css 后缀即可。不必减少，也不必增多。

2) Far-future Expires Header

配置 Web 服务器，更好的对待静态资源：

```
location ~ ^/assets/ {
  expires 1y;
  add_header Cache-Control public;

  add_header ETag "";
  break;
}
```

CDN 及异地静态资源

把静态资源放到其它服务器：

```
config.action_controller.asset_host = 'mycdnsubdomain.fictional-cdn.com'

config.action_controller.asset_host = ENV['CDN_HOST']
```

注意其影响：

```
<%= asset_path('smile.png') %>

http://mycdnsubdomain.fictional-cdn.com/assets/smile.png

# 仅作用于指定的静态资源
<%= asset_path 'image.png', host: 'mycdnsubdomain.fictional-cdn.com' %>
```

另，如果生产环境有图片，而开发环境没有，而自己又不想导图片。可以尝试用这种方式，在开发环境也显示图片。

两种引入方式

1）独立引入

```
config.assets.precompile += %w( site.css )
```

```
stylesheet_link_tag "site"
```

2）一起引入

```
# application.css

// = require_self
// = require 'site'
```

当然了，也可以直接把文件放到 public/assets/ 目录下，之后这些文件不受 Assets precompile 影响。

sprockets-rails

Railtie

以 Railtie 的形式引入，继承于 Rails::Railtie （所以，Rails::Railtie 提供的方法，它是可以用的）

主要任务包括，但不限于：

- 设置 config.assets.x (这里的 x 表示众多的配置项)
- 其它众多看不见的功能

上述配置，包括但不限于：

```
config.assets.precompile
config.assets.paths
config.assets.version
config.assets.prefix
config.assets.manifest
config.assets.digest
config.assets.debug
config.assets.compile
config.assets.configure
```

打开了 Rails 下的 Engine 和 Application

Engine 主要是加入路径：

```
paths["vendor/assets"]
paths["lib/assets"]
paths["app"]
paths["app/assets"]
```

Application 主要提供以下几个 config 项：

```
:assets
:assets_manifest
:precompiled_assets
```

如何引入

它是 gem，所以：

```
gem 'sprockets-rails', :require => 'sprockets/railtie'
```

Rake task

由 Task 文件完成。

```
rake assets:precompile
rake assets:clean
rake assets:clobber
```

命令

```
RAILS_ENV=production bin/rake assets:precompile
```

将 app/assets 存放普通的前端资源复制到 public/assets 目录。

Helper

包括但不限于：

```
include ActionView::Helpers::AssetUrlHelper
include ActionView::Helpers::AssetTagHelper
include Sprockets::Rails::Utils

VIEW_ACCESSORS = [:assets_environment, :assets_manifest,
                  :assets_precompile, :precompiled_assets,
                  :assets_prefix, :digest_assets, :debug_assets]
```

一些配置项

rails scaffold 或 rails g controller 时不再生成默认的 js, css 文件

```
config.generators do |g|
  g.assets false
end
```

显性配置 **css, js** 压缩器

```
config.assets.css_compressor = :yui
config.assets.js_compressor = :uglifier
```

```
config.assets.css_compressor = :yui
```

```
config.assets.css_compressor = :sass
```

JavaScript Compression

:closure, :uglifier 和 :yui

分别对应

closure-compiler, uglifier 和 yui-compressor gem.

```
config.assets.js_compressor = :uglifier
```

如何新增静态资源所在路径：

如何新增：

```
config.assets.paths << Rails.root.join("lib", "videoplayer", "fl
ash")
```

Runtime Error Checking

```
config.assets.raise_runtime_errors = false
```

没必要关掉吧。

Turning Debugging Off

```
config.assets.debug = false
```

没必要关掉吧。

Live Compilation

生产环境，实时编译。

开发、测试等环境，由 coffee-script and sass 实时编译。

反正我是不推荐：

```
config.assets.compile = true
```

生产环境 app/assets 下的文件已经预编译好，放到了 public/assets 下，直接使用即可。

CDNs and the Cache-Control Header

```
config.static_cache_control = "public, max-age=31536000"
```

Changing the assets Path

```
config.assets.prefix = "/some_other_path"
```

这也没必要吧。

Assets Cache Store

```
config.assets.cache_store = :memory_store
```

```
config.assets.cache_store = :memory_store, { size: 32.megabytes
}
```

```
config.assets.configure do |env|
  env.cache = ActiveSupport::Cache.lookup_store(:null_store)
end
```

一般情况下，也不会动这里哈~

ssets Pipeline 怎么关掉?

你可以在 confing/application.rb 中把他关掉：

```
config.assets.enabled = false
```

其它

查看所有 **assets** 目录

当使用第三方 gem 引入 assets 资源的时候，使用它可以让我们看到加入了哪些目录。

```
Rails.application.config.assets
```

> Note：它只管加载目录，想引入目录下面资源文件的话，还得自己或 gem 添加。

Deploy 的小技巧

1）本地编译 - 有好也有坏。反正我是不推荐。

2）如果 assets 沒有更新，就不要跑 precompile.

3) 有更新就在本地 precompile，然后再上传。

慎用 **require_tree**

```
# application.css
//= require_tree

# 相当于
//= require_tree .
```

意味着加载当前目录下的文件，并且递归加载其子目录下的文件。虽然省事了，这并不是好的实践。因为我们的文件、目录是会逐渐增多的。这就容易出现下列结果：

- 加载过多，导致性能慢
- 加载过多，导致混淆（原本没有必要加载的文件也加载了）

css & scss & sass

后缀名 .scss 意义 Sassy CSS
(还有一种后缀名为 .sass 的，区别是它严格缩进)

根据原有文件后缀名，选择不同的使用方式：

```
# 1 application.css
*= require font-awesome
```

```
# 2 application.css.scss
@import "font-awesome";
```

```
# 3 application.css.sass
@import font-awesome
```

检测 **assets** 是否挂了（存在）的命令

图片"rails.png"存在

```
Rails.application.assets.find_asset 'rails.png'
=> #<Sprockets::StaticAsset:0x3fed3aa004f8
pathname="/Users/kelby/appname/app/assets/images/rails.png",
mtime=2015-04-13 20:10:42 +0800, digest="3526faae1dacebb591431f0
054e8f33e">
```

图片"not-image.png"不存在

```
Rails.application.assets.find_asset 'not-image.png'
 => nil
```

没有 **JavaScript** 运行时

```
group :production do
  gem 'therubyracer'
end
```

但不推荐这么做，还是安装的好。

X-Sendfile Headers

Web 服务器支持的话，不妨一试：

```
# config.action_dispatch.x_sendfile_header = "X-Sendfile" # for
Apache
# config.action_dispatch.x_sendfile_header = 'X-Accel-Redirect'
# for NGINX
```

当你使用 `send_file` 等方法给客户端发送文件时，如果你开启了此特性，并且硬盘里有恰好又有此文件。那么，你的 Web服务器会忽略应用的响应数据，直接从硬盘读取文件并返回，使得速度更快。

不好的实践

以 controller 为单位加载资源

```
<%= javascript_include_tag params[:controller] %>
<%= stylesheet_link_tag params[:controller] %>
```

在我看来，上述方式并不好。

CSS 里引入图片、字体需注意

使用 css，特别是第三方样式的时候，注意查看是否引入了图片、字体，它们使用 url 引用，以及路径是否准确。

Debug

```
Rails.application.config.assets
```

查看资源情况。

它属于：

```
Rails.application.config.assets.class
=> Sprockets::Railtie::OrderedOptions
```

我们接触比较多的是其中的 `paths` 和 `precompile` 对应的数据。

Turbolinks 产生的原因

原因：

1. HTML 5 引进了 "History interface"
2. 原来的一个 request/response 需要传递整个页面（不论你有没有做缓存）
3. Assets Precompile 将整个静态资源打包，导致 css/js 太大
4. 借鉴了 PJAX

第 1 点补充：

```
interface History {
  readonly attribute long length;
  readonly attribute any state;
  void go(optional long delta);
  void back();
  void forward();
  void pushState(any data, DOMString title, optional DOMString?
url = null);
  void replaceState(any data, DOMString title, optional DOMStrin
g? url = null);
};
```

第 2 点补充：

```
request ->
response(title and body) <-

request ->
response(only body) <-
```

类似 AJAX 但 url 变了。

第 3 点补充：

有人说 Turblinks 就是为了解决 Assets Precompile 引起的问题。

第 4 点补充：

```
$.pjax({
  url: '/authors',
  container: '#main'
})
```

PJAX 是指定 dom，而 Turblinks 是整个 body.

Turbolinks 3

Ruby 层面：

```
Controller
    3 个回调方法：设置 X-XHR-Redirected-To，只对 GET 请求有效，某种情况的跨域是不允许的
    （用到了 X-XHR-Referer）
    重写 referer 方法(用到了 X-XHR-Referer )
    更改 redirect_to 计算规则（用到了 X-XHR-Referer ）
    重写 redirect_to 方法，某种情况下使用 Turbolinks.visit 代替
    重写 render 方法，某种情况下使用 Turbolinks.replace 代替
    render 和 redirect_to 可以额外处理参数：turbolinks，keep，change，flush
    （各个参数对应功能可以查看文档）
Router
    加上 _turbolinks_redirect_to（用到了 HTTP_X_XHR_REFERER ）
View
    加上 X-XHR-Referer

共引入标记
    cookies[:request_method]
    request.headers['X-XHR-Referer']
    request.headers["X-XHR-Referer"]
    session[:_turbolinks_redirect_to]
    response.headers['X-XHR-Redirected-To']
    env['HTTP_X_XHR_REFERER']
    env['rack.session'][:_turbolinks_redirect_to]
    controller.request.headers["X-XHR-Referer"]
```

JS 层面：

```
定义了几个变量
有哪些事件？
fetch 是如何实现的？
配置要不要缓存
fetch 核心部分之...（实际上是XMLHttpRequest对象；也用到了 X-XHR-Refere
r）
fetchHistory
cacheCurrentPage
如何缓存、如何清除缓存？
replace
fetch, replace 核心之...（如何才能快速且有效的替换？fetchReplacement）
window.history 记录历史链接、重定向链接、当前链接和状态
clone
各种乱七八糟的响应格式
过期
CSRFToken
核心之...createDocument
进度条
    也可以做得很复杂
进度条之... API
其它：ua，requestMethodIsSafe，browserSupportsTurbolinks，browserS
upportsCustomEvents
@Turbolinks 之 API
```

文档：

```
10 个事件
    整个 load 流程
    整个 replace 流程
举例说明可以如何使用这些事件
页面缓存（介绍，如何工作，如何配置，如何手动调用）
过滤缓存（配置要不要用即可，配置某个页面不使用）
进度条（好看，并且浏览器自带的进度条由于特性没有了）（配置要不要使用，配置样式，手动调用）
永久保存（配置即可），例如全局的侧边栏，并且只加载一次
配置某个地方的链接不使用 turboolinks，配置除 .html 外其它后缀的链接也加 turbolinks，
    配置不要所有 redirect_to 都使用 turbolinks 特性只在部分 Controller 单独引入使用
允许使用老的 JS 写法 jquery.turbolinks （引入即可）
配置是否检测 JS 或 CSS 为最近版本（假设我们重新部署，md5 变了，有时候 turbolinks 不知道）
配置 View 里的 script 只执行一次，还是每次都执行
手动调用，明确的指定使用 turbolinks
局部替换（也可以配合 data-turbolinks-temporary 一起使用）...
    客户端：要手动调用方法；服务端：同样的，要手动调用方法
配置（全局或针对某个请求），让浏览器不要缓存 turbolinks 请求
客户端 API (api 是 api，事件是事件，不要搞混了)
我们极度依赖 pushState （在这里，做为用户，我们就不要考虑了吧~~）
```

Turbolinks 补充

简单理解概念：把原来所有 GET 请求(包括链接、重定向、浏览器的后退)都通过 AJAX 来实现，并且 url 是变化的。

使用 turblinks 的话，注意：一，尽量从 AJAX 的角度出发写监听事件；二，尽量使用 turblinks 提供的监听事件。

Ruby 部分

重写了 **ActionController::Base** 里的 **redirect_to** 和 **render** 方法

```
ActionController::Base.ancestors.select{|a| a.to_s =~ /^Turbolinks/}
 => [Turbolinks::XHRHeaders, Turbolinks::Cookies,
     Turbolinks::XDomainBlocker, Turbolinks::Redirection]
```

render 使用的是 Turbolinks.replace
redirect_to 使用的是 Turbolinks.visit

处理 **:back** 这种情况，使用 **X-XHR-Referer** 标识

更改了 url_for 方法(会影响到 link_to 等调用到它的方法)

当 url_for 使用到 :back 参数时有用，它会把原来 header 里的 HTTP_REFERER 改为 HTTP_X_XHR_REFERER

原因：Turbolinks 获取内容的时候使用 XMLHttpRequest 发起请求(简称 XHR)

带来的问题：XHR 请求的 HTTP 头 Referer 并不正确

解决办法：添加了 X-XHR-Referer 头，后端程序获取 Referer 的时候，需要先取 X-XHR-Referer。

```
module XHRUrlFor
  def self.included(base)
    base.alias_method_chain :url_for, :xhr_referer
  end

  def url_for_with_xhr_referer(options = {})
    options = (controller.request.headers["X-XHR-Referer"] || op
tions) if options == :back
    url_for_without_xhr_referer options
  end
end

# 辅助方法
def referer
  self.headers['X-XHR-Referer'] || super # super 就是 self.header
s['Referer']
end
alias referrer referer
```

覆盖了 **Redirect** 的 **call** 方法，修复 **redirect_to** 里的 **bug**

```
def call_with_turbolinks(env)
  status, headers, body = call_without_turbolinks(env)

  if env['rack.session'] && env['HTTP_X_XHR_REFERER']
    env['rack.session'][:_turbolinks_redirect_to] = headers['Loc
ation']
  end

  [status, headers, body]
end
alias_method_chain :call, :turbolinks
```

注意：上面设置了头 X-XHR-Redirected-To

其它几点

一，3 个钩子

```
before_action :set_xhr_redirected_to, :set_request_method_cookie
after_action :abort_xdomain_redirect
```

1) 因为 trublinks 改变了原来 redirect_to 的行为，带来了新的问题，然后它又提供了解决办法：

```
  def set_xhr_redirected_to
    if session && session[:_turbolinks_redirect_to]
      response.headers['X-XHR-Redirected-To'] = session.delete :
_turbolinks_redirect_to
    end
  end
```

对应 redirect_to :back 这种情况，使用 X-XHR-Referer 标识；用 X-XHR-Redirected-To 记录重定向的目标地址。

2) 它怎么记录是非 GET 请求的呢？用 cookies 记录。

```
  def set_request_method_cookie
    if request.get?
      cookies.delete(:request_method)
    else
      cookies[:request_method] = request.request_method
    end
  end
```

对于非 GET 请求，turblinks 是不起作用的。

3) 跨域重定向：

```
  def abort_xdomain_redirect
    to_uri = response.headers['Location']
    current = request.headers['X-XHR-Referer']
    unless to_uri.blank? || current.blank? || same_origin?(curre
nt, to_uri)
      self.status = 403
    end
  rescue URI::InvalidURIError
  end
```

某种情况下，禁止访问。

JS 部分

所有对外 API

```
Public API
   Turbolinks.visit(url)
   Turbolinks.pagesCached()
   Turbolinks.pagesCached(20)
   Turbolinks.cacheCurrentPage()
   Turbolinks.enableTransitionCache()
   Turbolinks.disableRequestCaching()
   Turbolinks.ProgressBar.enable()
   Turbolinks.ProgressBar.disable()
   Turbolinks.ProgressBar.start()
   Turbolinks.ProgressBar.advanceTo(80)
   Turbolinks.ProgressBar.done()
   Turbolinks.allowLinkExtensions('md')
   Turbolinks.supported
   Turbolinks.EVENTS

# 去除重复的有：
  visit,
  replace,
  pagesCached,
  cacheCurrentPage,
  enableTransitionCache,
  disableRequestCaching,
  ProgressBar: ProgressBarAPI (包括：enable, disable, start, addv
anceTo, done),
  allowLinkExtensions: Link.allowExtensions,
  supported: browserSupportsTurbolinks,
  EVENTS: clone(EVENTS)
```

配置项：

cacheSize 默认缓存 10 个页面

transitionCacheEnabled 事务缓存？默认为 false

requestCachingEnabled 请求缓存？默认为 true

progressBar 进度条

currentState 当前状态
loadedAssets 加载资源

referer 后退链接

xhr 异步请求

所有事件：

```
EVENTS =
  BEFORE_CHANGE:  'page:before-change'
  FETCH:          'page:fetch'
  RECEIVE:        'page:receive'
  CHANGE:         'page:change'
  UPDATE:         'page:update'
  LOAD:           'page:load'
  RESTORE:        'page:restore'
  BEFORE_UNLOAD:  'page:before-unload'
  EXPIRE:         'page:expire'
```

具体做了什么事，如何实现，可以查看对应源代码，在此不做讨论。

一些特性

Transition Cache

前面不是说过 turblinks 默认可以缓存 10 个页面吗？
现在问题来了：如果我要访问一个已经存在缓存里的页面，它是"直接从缓存里取"，还是"改为了 AJAX 实现重新从服务端取"呢？

默认还是从"改为了 AJAX 实现重新从服务端取"。不过，你可以通过以下配置，让它"直接从缓存里取"：

```
Turbolinks.enableTransitionCache();
```

Partial replacements

你可以使用

```
Turbolinks.visit(path, options)
```

或

```
Turbolinks.replace(html, options)
```

实现局部替换 HTML 文档内容。

坑及策略

事件的改变

1) 点击链接：

```
page:before-change # 链接刚刚被点击，turblinks 刚准备起作用的状态
page:fetch         # 从服务端获取内容
page:receive       # 已经收到服务器返回内容，但还没处理(去掉 header 等)
的状态
page:before-unload # 已经处理完成，但还没替换的状态
page:change        # 已切换到新页面的状态
page:load          # 整个点击彻底的完成
```

2) 而使用浏览器历史返回上一页：

```
page:before-unload # 页面已经从缓存里取出，准备切换
page:change        # 已切换到新页面的状态
page:restore       # 整个回退彻底的完成
```

3) 其它：

```
# 到目前为止，我们就有两类 AJAX 请求了。
# 一是 turblink 转换过来的 AJAX，二是我们编写的、真正的 AJAX.
# 使用下面这个状态，可以同时监听上述两种 AJAX 引起的页面变化。
page:update

# 手动缓存当前页面
Turbolinks.cacheCurrentPage();

# 页面缓存数量达到上限时，触发它：
page:expire

# 效果和 fetch 一样，只不过多了个判断。
# 如果浏览器支持 turblinks 那么使用 turblinks 的 fetch.
# 如果浏览器不支持 turblinks 那么使用普通的访问
Turbolinks.visit

# 局部替换 HTML 文档内容
Turbolinks.replace
```

JS 要如何更改使用？

jquery.turbolinks 实践，请按照 **AJAX** 的方式写 **JS** 代码。

```
/* 不要这么写 */
$(function() {
  $(document).on('click', 'button', function() { ... })
});

/* 而是这么写 */
$(document).on('click', 'button', function() { ... })
```

和

```
// 不要这么写
$(document).on('ready', function () { /* ... */ });

// 而是这么写
$(document).ready(function () { /* ... */ });
$(function () { /* ... */ });
```

怎样防止浏览器缓存 **Turbolinks** 请求:

```
# 全局
Turbolinks.disableRequestCaching()

# 每个请求
Turbolinks.visit url, cacheRequest: false
```

像 jQuery.ajax(url, cache: false), 会被追加 "_#{timestamp}" 到 GET 请求参数里。

如何移除

```
# Gemfile

# 注释这一行:
gem 'turbolinks'
```

```
# app/assets/javascripts/application.js

# 移除这一行:
//= require turbolinks
```

如果有 **js** 逻辑绑定了**DOMContentLoaded** 事件，就需要为 **Turbolinks** 做兼容语法或历史遗留问题，可以使用以下方法：

```
# Gemfile:
gem 'jquery-turbolinks'
```

```
# Add it to your JavaScript manifest file, in this order:
//= require jquery
//= require jquery.turbolinks
//= require jquery_ujs
//
// ... your other scripts here ...
//
//= require turbolinks
```

几个 **data-*** 属性

```
data-no-transition-cache

data-turbolinks-permanent

data-no-turbolink

data-method, data-remote, or data-confirm

data-turbolinks-track

data-turbolinks-eval

data-turbolinks-temporary
```

原来的 **DOM** 操作怎么办？并且它们是非幂等的

DOM 操作最好是幂等。如果不是幂等的，你可以使用使用 page:load 而不是 page:change

```
// using jQuery for simplicity

$(document).on("ready page:load", nonIdempotentFunction);
```

其它类和对象

Click 管理着链接和点击事件，触发时会进行检查。如果符合 trublinks 则用 AJAX 执行请求，否则改用普通的请求。

ProgressBar 进度条。对应 class name 为 turbolinks-progress-bar

ProgressBarAPI 管理进度条可用的接口。有：

```
enable
disable

start
done

advanceTo
```

CSRFToken 管理 csrf token，提供接口：

```
get
update
```

Link 对于已经存在的链接，进行一些特殊处理。 比如，前面提到过的，有的链接不能使用 turblinks

各种浏览器不同引起的问题

如何替换链接？

由 JS 实现。

如何防止丢失 Referer ？

压根就没有丢失，只是原来的不能用，用新的 X-XHR-Referer

参考：

Turbolinks
jquery.turbolinks

Turbolinks 5

比 Turbolinks 3 好用，代码事整洁，文档也更友好。如果你之前直接关闭了 Turbolinks 的话，现在建议你尝试一下。

两方面： 一、Ruby 层面 二、JS 层面

访问方式： GET 第一次访问页面 GET 第二次访问页面 刷新页面 重定向到页面 浏览器提供的"后退" 浏览器提供的"前进"

原理： ajax 请求，改变 url，丢掉 head 部分、直接替换 body 部分。 window 和 document 保持不变。

底层第三方技术： 重点： HTML5 History API

一般： Window.requestAnimationFrame XMLHttpRequest

其它： MutationObserver Custom Elements

工作方式： Turbolinks.xxx 直接调用 data-turbolinks-xxx 间接声明

特殊情况，特殊处理： 一般都是通过 turbolinks-xxx data-turbolinks-xxx 等方式声明当时是特殊情况，要求特殊处理。

data-turbolinks data-turbolinks-action data-turbolinks-eval data-turbolinks-preview data-turbolinks-permanent data-turbolinks-track

turbolinks-cache-control turbolinks-progress-bar turbolinks-root

生命周期： turbolinks:click turbolinks:before-visit turbolinks:visit turbolinks:request-start turbolinks:request-end turbolinks:before-cache turbolinks:before-render turbolinks:render turbolinks:load

内部实现两种访问方式： 直接从缓存里拿 从服务器响应里拿

内部关键术语： Turbolinks-Location

部分 **API**： Turbolinks.visit Turbolinks.clearCache Turbolinks.supported

问题： 重复执行 不执行 首次执行，后续不执行 document.read 等老的写法 页面引入 script 等。

建议： 先看官方文档（新、权威） 再看源代码 对比着看相关参考文章

Testing

TODO

Active Support

包含了：Test Case、Assertions、Declarative、Isolation、Setup 和 Teardown 相关类方法、Time Helpers、Log Subscriber Test Helper 等部分。

Test Case

```
assert_nothing_raised

# 为了向之前的 test/unit 兼容
alias :assert_raise :assert_raises
alias :assert_not_empty :refute_empty
alias :assert_not_equal :refute_equal
alias :assert_not_in_delta :refute_in_delta
alias :assert_not_in_epsilon :refute_in_epsilon
alias :assert_not_includes :refute_includes
alias :assert_not_instance_of :refute_instance_of
alias :assert_not_kind_of :refute_kind_of
alias :assert_no_match :refute_match
alias :assert_not_nil :refute_nil
alias :assert_not_operator :refute_operator
alias :assert_not_predicate :refute_predicate
alias :assert_not_respond_to :refute_respond_to
alias :assert_not_same :refute_same
```

```
test_order
test_order=
```

test_order 支持：

```
ActiveSupport::TestCase.test_order # => :sorted

:random   # 随机
:parallel # 并行
:sorted   # 按字母顺序
:alpha    # 按字母顺序
```

Assertions

```
assert_difference
assert_no_difference

assert_not
```

Declarative

```
test
```

Isolation

类方法：

```
forking_env?
```

实例方法：

```
run
```

Forking：

```
run_in_isolation
```

Subprocess：

```
run_in_isolation
```

Setup & Teardown 相关类方法

```
setup
teardown
```

使用举例：

```
class ExampleTest < ActiveSupport::TestCase
  setup do
    # ...
  end

  teardown do
    # ...
  end
end
```

文档没写，但还有：

```
before_setup
after_teardown
```

Time Helpers

```
travel
travel_back
travel_to
```

Log Subscriber - Test Helper

```
set_logger

setup
teardown

wait
```

使用举例：

```
class SyncLogSubscriberTest < ActiveSupport::TestCase
  include ActiveSupport::LogSubscriber::TestHelper

  def setup
    ActiveRecord::LogSubscriber.attach_to(:active_record)
  end

  def test_basic_query_logging
    Developer.all.to_a
    wait # <-- 这个
    assert_equal 1, @logger.logged(:debug).size
    assert_match(/Developer Load/, @logger.logged(:debug).last)
    assert_match(/SELECT \* FROM "developers"/, @logger.logged(:
debug).last)
  end
end
```

File Fixtures

```
file_fixture
```

Constant Lookup

```
determine_constant_from_test_name
```

Deprecation

```
assert_deprecated
assert_not_deprecated
```

Active Job

Test Helper

实例方法：

```
assert_enqueued_with
assert_performed_with

assert_enqueued_jobs
assert_performed_jobs

assert_no_enqueued_jobs
assert_no_performed_jobs
```

```
perform_enqueued_jobs
```

以及：

```
delegate :enqueued_jobs, :enqueued_jobs=, :performed_jobs, :perf
ormed_jobs=,
         to: :queue_adapter
```

和其它测试一样，你也可以自己编写前后过滤器：

```
before_setup

after_teardown
```

默认测试环境下使用 queue_adapter 是 test，类似：

```
Rails.application.config.active_job.queue_adapter = :test
```

Action Mailer

Test Helper

默认 Rails 提供两个 helper 方法用于测试：

方法	解释
assert_emails	断言已经发送的邮件数
assert_no_emails	断言没有邮件发送出去(可用 assert_emails 0 代替)
assert_enqueued_emails	断言邮件已进队列
assert_no_enqueued_emails	断言邮件不在队列里

assert_emails 和 assert_no_emails 两者本质都是封装 assert_equal.

~~Behavior~~

类方法：

```
determine_default_mailer

mailer_class

tests
```

~~实例方法：~~

```
initialize_test_deliveries

restore_delivery_method
restore_test_deliveries

set_delivery_method
set_expected_mail
```

除通常的测试方法外，还有 `deliveries` 方法获取已经发送的邮件实例。

使用举例:

```
last_email = ActionMailer::Base.deliveries.last

expect(last_email.to).to eq ['test@example.com']
expect(last_email.subject).to have_content 'Welcome'

email = UserMailer.confirmation(user.id).deliver_now

assert ActionMailer::Base.deliveries.any?
assert_equal [user.email], email.to
```

Action Controller

Test Case Behavior

实例方法：

```
get
post
patch
put
delete
head

xhr & xml_http_request

process
```

> Rails 5 新增 xhr 虽然是语法糖，但用起来很舒心，不是吗？

Live Test Response

```
success? & successful?
missing? & not_found?
error? & server_error?
```

Action Dispatch

Assertions

实例方法：

```
html_document
```

Response Assertions：

```
assert_redirected_to
assert_response
```

Routing Assertions：

```
assert_generates
assert_recognizes
assert_routing

with_routing
```

Test Process

```
# Rails 5 已废除 assigns
assigns

cookies
flash
session

fixture_file_upload
redirect_to_url
```

Test Request

```
accept=
action=

cookies & rack_cookies

host=

if_modified_since=
if_none_match=

path=
port=

remote_addr=
request_method=
request_uri=

user_agent=
```

Test Response

```
from_response

# response successful?
success? & successful?

# URL not found?
missing? & not_found?

# redirected?
redirect? & redirection?

# a server-side error?
error? & server_error?
```

Integration Test

类方法：

```
app
app=
```

实例方法：

```
app

document_root_element

url_options
```

Request Helpers：

```
delete

delete_via_redirect

follow_redirect!

get

get_via_redirect

head

patch

patch_via_redirect

post

post_via_redirect

put

put_via_redirect

request_via_redirect

xhr & xml_http_request
```

Session：

```
cookies

host

https?
https!

reset!

url_options
```

Runner：

```
app

default_url_options
default_url_options=

open_session

reset!
```

Action View

Behavior

实例方法：

```
config

render
rendered_views

setup_with_controller
```

类方法：

```
determine_default_helper_class

helper_class
helper_method

tests
```

Rendered Views Collection：

```
add
locals_for

rendered_views

view_rendered?
```

Test Controller

```
controller_path=
```

Generators

Assertions

```
assert_class_method
assert_instance_method & assert_method

assert_file & assert_directory
assert_no_file & assert_no_directory

assert_migration
assert_no_migration

assert_field_type
assert_field_default_value
```

Behaviour

类方法：

```
arguments

destination

tests
```

实例方法：

```
create_generated_attribute

generator

run_generator
```

其它实例方法：

```
capture
```

~~Setup And Teardown~~

```
setup
teardown
```

> Note: 上述模块们于 generators/testing/ 目录下。

rails-dom-testing

从 ActionView 分离出来的 gem，主要由以下两部分构成：

Dom Assertions

```
assert_dom_equal '<h1>Lingua França</h1>', '<h1>Lingua França</h1>'

assert_dom_not_equal '<h1>Portuguese</h1>', '<h1>Danish</h1>'
```

Selector Assertions

```
# implicitly selects from the document_root_element
css_select '.hello' # => Nokogiri::XML::NodeSet of elements with hello class

# select from a supplied node. assert_select asserts elements exist.
assert_select document_root_element.at('.hello'), '.goodbye'

# elements in CDATA encoded sections can also be selected
assert_select_encoded '#out-of-your-element'

# assert elements within an html email exists
assert_select_email '#you-got-mail'
```

MiniTest

链接 minitest

Spec DSL

实例方法：

```
it

let
subject

before
after
```

`it` 和 def test_x 类似。一般的，每一个测试案例对应着一个测试方法。可用 it 定义它们。

`let` 和 def 定义方法类似，只不过这里方法内容始终是 block. 有延迟加载的作用。

`subject` 和 let 类似，但它没有名字。(其实它是有名字的，始终是 :subject)

`before` 前置过滤器，在每一个 it 之前执行。(名字无所谓，只是起标识作用而矣)

`after` 后置过滤器，在每一个 it 之后执行。(名字无所谓，只是起标识作用而矣)

> before、after 方法和 RSpec 里的 setup、teardown 方法有对应关系。

除上述方法外，还有：

```
spec_type
register_spec_type

children
```

链接 MiniTest::Spec::DSL

Assertions

实例方法：

```
assert
assert_empty
assert_equal
assert_in_delta
assert_in_epsilon
assert_includes
assert_instance_of
assert_kind_of
assert_match
assert_nil
assert_operator
assert_output
assert_predicate
assert_raises
assert_respond_to
assert_same
assert_send
assert_silent
assert_throws

# refute 和 assert 意思正好反着来

refute
refute_empty
refute_equal
refute_in_delta
refute_in_epsilon
refute_includes
refute_instance_of
refute_kind_of
refute_match
refute_nil
refute_operator
refute_predicate
refute_respond_to
refute_same
```

```
pass

skip

flunk
```

除上述方法外，还有：

```
capture_io
capture_subprocess_io

diff

exception_details

message

mu_pp # 表示 human-readable pretty_print
mu_pp_for_diff


skipped?

synchronize
```

类方法：

```
diff
diff=
```

链接 MiniTest::Assertions

Mock

类方法：

```
expect
verify
```

`expect` 第一个参数为方法名，第二个参数为执行此方法后返回的结果，第三个参数表示传递给此方法的参数，也可以传递一个 block.

使用举例：

```
# Expect that method name is called, optionally with args or a b
lk, and returns retval.
m = Minitest::Mock.new
m.expect(:raiser, nil) do |args|
  raise RuntimeError, "this code path triggers an exception"
end
```

我们可以使用 Minitest::Mock.new 模拟一些不能(或不希望)直接调用的对象。

```
user = Minitest::Mock.new
user.expect(:delete, true) # returns true, expects no args

UserDestoyer.new.delete_user(user)

assert user.verify
```

`verify` mock 出来的对象有时候并不是我们想要的(计算机不够聪明)，verify 验证是否真的执行了上述方法。RSpec 在这点上使用起来更简洁，但也相差不大。

链接 MiniTest::Mock

包含 TestCase、Lifecycle Hooks 和 Guard 三部分。

TestCase

实例方法：

```
setup
teardown
```

除上述方法外，还有：

```
io
io?

run

passed?
```

类方法：

```
i_suck_and_my_tests_are_order_dependent! # 让测试按照顺序执行！

make_my_diffs_pretty! # 错误日志格式化，更漂亮，但会降慢速度

parallelize_me! # 并行执行测试

benchmark_suites

bench_exp
bench_linear
bench_range
```

链接 MiniTest::Unit::TestCase

Lifecycle Hooks

实例方法：

```
after_setup
after_teardown

before_setup
before_teardown
```

链接 MiniTest::Unit::LifecycleHooks

Guard

类方法：

```
maglev?
```

实例方法：

```
jruby?
mri?
rubinius?
windows?
```

TestCase 以 mixed 的方式引入此模块，所以可以以类方法的形式调用这里的实例方法。

链接 MiniTest::Unit::Guard

~~Expectations~~

可以用 assert_x 和 refute_x 来代替这里的方法，文档和实现都一样。

实例方法：

```
# 以下方法和 RSpec 对应，可以使用 assert_x 代替
must_be
must_be_close_to
must_be_empty
must_be_instance_of # 范围比 kind_of 小，子类的实例不行
must_be_kind_of     # 范围比 instance_of 大，子类的实例也行
must_be_nil
must_be_same_as
must_be_silent
must_be_within_delta
must_be_within_epsilon
must_equal
must_include
must_match
must_output
must_raise
must_respond_to
must_send
must_throw

# must_x 和 wont_x 意思正好反着来

# 以下方法和 RSpec 对应，可以使用 refute_x 代替
wont_be
wont_be_close_to
wont_be_empty
wont_be_instance_of
wont_be_kind_of
wont_be_nil
wont_be_same_as
wont_be_within_delta
wont_be_within_epsilon
wont_equal
wont_include
wont_match
wont_respond_to
```

链接 MiniTest::Expectations

其它

其它多个类或模块，统一在此列举。

Kernel

很常用的方法：

```
describe
```

Object

常用的方法：

```
stub
```

保存调用此方法的值，下次再调用此方法，直接使用该值，不用重新计算生成。

举例：

```
DateTime.now
# => 2014-11-17T22:16:15+08:00

DateTime.stub :now, (Date.today.to_date - 14) do
  puts DateTime.now
  # ...
  puts DateTime.now
end

# => 2014-11-03
# => 2014-11-03
```

Benchmark

已经 mixed 进 TestCase.

类方法：

```
bench_exp
bench_linear
bench_range
```

实例方法：

```
assert_performance
assert_performance_constant
assert_performance_exponential
assert_performance_linear
assert_performance_logarithmic
assert_performance_power

fit_error
fit_exponential
fit_linear
fit_logarithmic
fit_power

sigma

validation_for_fit
```

Bench Spec

类方法：

```
bench
bench_performance_constant
bench_performance_exponential
bench_performance_linear
bench_range
```

Abstract Reporter

实例方法：

```
passed?

record
report

start
```

Assertion

实例方法：

```
location
```

Factory Girl

可用于构建 record 对象的方法

构建单个对象：

```
build
create
attributes_for
build_stubbed
```

使用举例：

```
build(:completed_order)

create(:post) do |post|
  create(:comment, post: post)
end

attributes_for(:post, title: "I love Ruby!")

build_stubbed(:user, :admin, :male, name: "John Doe")
```

构建多个对象：

```
build_list
create_list
attributes_for_list
build_stubbed_list
```

使用举例：

```
build_list(:completed_order, 2)
create_list(:completed_order, 2)

attributes_for_list(:post, 4, title: "I love Ruby!")

build_stubbed_list(:user, 15, :admin, :male, name: "John Doe")
```

链接 FactoryGirl Syntax Methods

Getting Started

Linting Factories

类方法：

```
lint
```

Defining factories

类方法：

```
define
```

实例方法：

```
factory
```

(定义 factory.)

Lazy Attributes

定义属性时，大括号 `{}` 的使用。

Aliases

调用 factory 时，可选参数 `:aliases` 的使用(给 factory 起外号)。

Dependent Attributes

`{}` 大括号里面求值(拼接字符串)。

Transient Attributes

```
transient
```

(在定义 factory 的代码内同时定义方法，之后使用到)

Associations

```
association
```

以及它的可选参数 `:factory` 和 `:strategy`

(关联对象的 factory.)

Inheritance

嵌套使用 `factory` 方法。

或 factory 的可选参数 `:parent`

(继承已有的 factory.)

Sequences

```
sequence

generate
```

(按顺序生成 factory 的属性内容)

Traits

定义：用 `trait` 进行定义。

调用：factory 的可选参数 `:traits` 或直接调用。

(避免重复代码)

Callbacks

默认已经有：

```
after(:build)
before(:create)
after(:create)
after(:stub)
```

(创建 factory 时的回调)

自定义：

```
callback
```

Modifying factories

类方法：

```
modify
```

(更改已有 factory. 场景：在 rails c 里检验)

Building or Creating Multiple Records

```
build_list
create_list

build_pair
create_pair
```

(批量创建 factory 对象)

Custom Construction

```
initialize_with

attributes
```

需要自定义相关类及方法。

(Model 里使用了 initialize 并设置了实例变量)

Custom Strategies

类方法：

```
register_strategy
```

需要自定义相关类及方法。

(自定义 factory 对象的生成规则)

Custom Callbacks

```
callback
```

使用"Custom Strategies"会自动添加上述默认的 Callbacks.

Custom Methods to Persist Objects

```
to_create

skip_create
```

(重新定义并调用"保存方法"；创建 factory 对象的时候跳过"保存")

重新加载 factory

类方法：

```
find_definitions
```

链接 Getting Started

后记

一些建议：

- 看新代码。Rails 版本更新太快，无论是新手还是老手，都建议从新的开始学习。不看新代码，之前在旧版本里遇到的问题或解决方案，或许新版本已经有了更好的解决方案，并且还新增了一些新特性可以使用。
- 各个 release note 一定要认真看！你不需要知道整个改进过程，但是要大致了解。
- 不用想着什么都学会了，才动手。几乎不可能(也没必要)全部知识都能学会，并且知识从实践中来，只有动手做过才能掌握、熟练。
- 知识点尽量全面。有时候我们重复轮子，既花时间又不够理想，却不知有些方法 Rails 已经内置，直接用即可。
- 不用完全迷信'更优雅'。框架，顾名思义，提供了方便也会有限制。当实际需求需要的时候，没必要什么事都往框架提供的内置方法去套。应该优先考虑完成任务，及后续阅读、维护。

目标：

- 尽量简洁。
- 尽量把晦涩、难懂的东西讲明白。
- 帮助人理解 Rails 5 整体架构，做他们阅读源代码的带路人。
- 对 API 的补充。
- 理解 Rails 的优雅之处。(有些东西，我们自己也可以做，但 Rails 提供了，而且更优雅；我们需要理解它，然后使用它)
- 使用和原理相结合。尽量两者兼顾，把握好度。
- 重要的是解决问题的能力，而非专家。
- 没有最好的解决方案。
- 当你不确定时，不妨多敲几行代码，以防万一。

目标读者: 普通 Rails Web 开发者。